Agility Compiler manual

For Agility 1.2

Celoxica and the Celoxica logo trademarks of Celoxica Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous development and improvement. All particulars of the product and its use contained in this document are given by Celoxica Limited in good faith. However, all warranties implied or express, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Celoxica Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any incorrect use of the product.

The information contained herein is subject to change without notice and is for general guidance only.

Copyright © 1991 - 2006 Celoxica Limited. All rights reserved.

Authors: RG

Document number: 17046-1.2-2

Customer Support at http://www.celoxica.com/support/

Celoxica in Europe	Celoxica in Japan	Celoxica in the Americas
T: +44 (0) 1235 863 656	T: +81 (0) 45 331 0218	T: +1 800 570 7004
E: sales.emea@celoxica.com	E: sales.japan@celoxica.com	E: sales.america@celoxica.com

License for Scintilla and SciTE

Copyright 1998-2003 by Neil Hodgson <neilh@scintilla.org>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in
supporting documentation.

NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
OR PERFORMANCE OF THIS SOFTWARE.

1 CELOXICA AGILITY USER MANUAL 13
1.1 PROJECT DEVELOPMENT SEQUENCE 13
1.1.1 Agility workflow 14
1.1.2 Agility walk-through 14

2 CELOXICA SYSTEMC SYNTHESIZABLE SUBSET 31
2.1 AGILITY SYSTEMC SYNTHESIZABLE SUBSET OVERVIEW 31
2.1.1 The synthesizable subset 32
2.2 MODULES 32
2.2.1 The SC_MODULE macro 33
2.2.2 Hierarchical modules 35
2.2.3 Port types 37
2.2.4 Channels 42
2.2.5 Processes 43
2.3 DATATYPES 52
2.3.1 C++ datatypes 52
2.3.2 SystemC numeric types 59
2.3.3 SystemC bit vector and logic vector types 64
2.3.4 Summary 65
2.4 TYPE PROMOTION AND CONVERSIONS 65
2.5 VARIABLES 66
2.6 EXPRESSIONS AND STATEMENTS 67
2.6.1 Expressions 67
2.6.2 Statements 69
2.7 FUNCTIONS 70
2.8 NAMESPACES 71
2.9 EXCEPTIONS 71
2.10 TRANSACTION-LEVEL MODELLING 71
2.11 AGILITY EXTENSIONS 71
2.11.1 Agility-specific types and functions 72
2.11.2 Black boxes 72
2.11.3 Global asynchronous reset 74
2.11.4 Initialization 75
2.11.5 Assertions 78
2.11.6 Signal with default 80
2.12 LANGUAGE FEATURES AND RESTRICTIONS SUMMARY 83
2.12.1 Supported SystemC language features 83
2.12.2 Supported SystemC/C++ types 83

3 MATHS LIBRARY 85
3.1 AGILITY MATHS LIBRARY FUNCTIONS 85
3.2 HEADER 85
3.3 DOUBLE PRECISION ANSI C FUNCTIONS 85
3.4 SINGLE PRECISION ANSI C FUNCTIONS 87

4 AGILITY SYNTHESIS GUIDE 90
4.1 AGILITY DESIGN FLOW 90
4.1.1 Design flow stages 90
4.2 CONSTRAINTS 91
4.2.1 Synthesis constraints 92
4.2.2 Output constraints 114
4.3 AGILITY SYNTHESIS 115
4.3.1 Compile-time evaluation 116

4.3.2 Loop unrolling ... 116
4.3.3 Compile-time recursive definitions ... 118
4.3.4 Function inlining ... 119
4.3.5 Pointer resolution ... 119
4.3.6 High-level synthesis ... 120

5 LOOP OPTIMISATION ... 122
5.1 AGILITY OUTPUT FORMATS ... 130
5.1.1 EDIF ... 130
5.1.2 VHDL ... 132
5.1.3 Verilog ... 135
5.1.4 RTL SystemC ... 139
5.1.5 HDL output formats ... 140
5.1.6 HDL output references ... 141
5.2 FACILITIES FOR REVIEWING SYNTHESIS RESULTS ... 141
5.2.1 Logic estimator ... 141
5.2.2 Report files ... 144
5.3 PREPARING CODE FOR SIMULATION ... 147
5.3.1 Simulating clocks correctly ... 147
5.4 TARGETING HARDWARE ... 147
5.4.1 Preparing to target hardware using Agility ... 147
5.4.2 Converting interfaces to hardware ... 147
5.4.3 Top-level port synthesis ... 147
5.4.4 Targeting a particular synthesis tool ... 148
5.4.5 Technology mapping ... 148
5.4.6 Targeting hardware via EDIF ... 150
5.4.7 Targeting hardware via Verilog ... 151
5.4.8 Targeting hardware via VHDL ... 152

6 USING AGILITY ... 153

7 AGILITY IDE GUIDE ... 154
7.1 STARTING THE IDE ... 154
7.2 IDE OVERVIEW ... 154
7.2.1 Workspace window ... 155
7.3 CREATING A PROJECT ... 156
7.4 ADDING FILES TO A PROJECT ... 156
7.5 REMOVING FILES FROM A PROJECT ... 156
7.6 ADDING A PROJECT TO A WORKSPACE ... 156
7.7 CONFIGURING A PROJECT ... 157
7.7.1 Modifying configurations ... 157
7.8 VIEWING AND EDITING PROJECT SETTINGS ... 158
7.8.1 Changing the configuration displayed ... 159
7.9 SPECIFYING THE INCLUDE PATH ... 160
7.9.1 Using file and directory macros ... 160
7.10 CUSTOM BUILD STEPS ... 161
7.10.1 Project custom build steps ... 162
7.10.2 Running user specified tools instead of Agility compiler ... 162
7.11 BUILDING A PROJECT ... 163
7.11.1 Setting the active project and the active configuration ... 163
7.12 USING THE REPORT VIEW ... 164
7.13 THE CODE EDITOR ... 164
7.14 FIND IN FILES ... 164
7.15 SIMULATING WITH THE OSCI SYSTEMC LIBRARIES ... 164

7.16 THE SIMULATION PATH 165

8 CELOXICA CDFGVIEWER 166
8.1 AGILITY CDFGVIEWER OVERVIEW 166
8.1.1 What is a CDFG? 166
8.1.2 The Agility CDFG 167
8.1.3 Groups 167
8.1.4 Nodes 167
8.1.5 Edges 168
8.2 ELEMENTS OF THE AGILITY CDFGVIEWER 168
8.2.1 Group rendering 168
8.2.2 Node rendering 169
8.2.3 Edge rendering 171
8.3 USING THE CDFGVIEWER 171
8.3.1 CDFGViewer display options 171
8.3.2 Creating a file for the CDFGViewer 173
8.3.3 Viewing a CDFGViewer file 173
8.3.4 CDFGViewer Module Call Graph 176
8.4 APPENDIX: COMPLETE LIST OF NODES 177

9 AGILITY COMMAND-LINE COMPILER REFERENCE 183
9.1 USING THE COMMAND-LINE COMPILER 183
9.1.1 Command-line options 183
9.1.2 Example: Command line 188
9.2 OPTION DEPENDENCIES 188

10 AGILITY EXAMPLES 190

11 AGILITY EXAMPLES SUPPLIED 190

12 AGILITY EXAMPLE: ADDONE 191
12.1 BUILDING THE ADDONE EXAMPLE FOR SIMULATION 191
12.2 SIMULATING THE ADDONE EXAMPLE 191
12.3 BUILDING THE ADDONE EXAMPLE FOR HARDWARE 191

13 AGILITY EXAMPLE: SC_METHODS 192
13.1 BUILDING SC_METHODS FOR SIMULATION 192
13.2 SIMULATING THE SC_METHODS EXAMPLE 192
13.3 BUILDING THE SC_METHODS EXAMPLE FOR HARDWARE 192

14 AGILITY EXAMPLE: LED 193
14.1 BUILDING THE LED EXAMPLE FOR SIMULATION 193
14.2 SIMULATING THE LED EXAMPLE 193
14.3 BUILDING AND RUNNING THE LED EXAMPLE FOR HARDWARE 194

15 AGILITY EXAMPLE: SEVENSEG 194
15.1 BUILDING THE SEVENSEG EXAMPLE FOR SIMULATION 194
15.2 SIMULATING THE SEVENSEG EXAMPLE 195
15.3 BUILDING AND RUNNING THE SEVENSEG EXAMPLE FOR HARDWARE 195

16 AGILITY EXAMPLE: COMPLEX 195
16.1 BUILDING THE COMPLEX EXAMPLE FOR SIMULATION 195
16.2 SIMULATING THE COMPLEX EXAMPLE 196

16.3 BUILDING THE COMPLEX EXAMPLE FOR HARDWARE 196

17 EXAMPLES OF AGILITY FEATURES 196
17.1 AGILITY MULTICLOCK EXAMPLE 196
17.1.1 Building the multiclock example for simulation 196
17.1.2 Simulating the multiclock example 197
17.1.3 Building and running the multiclock example for hardware 197
17.2 AGILITY EXAMPLE: BLACKBOX 197
17.2.1 Building the blackbox example for simulation 198
17.2.2 Simulating the blackbox example 198
17.2.3 Building the blackbox example for hardware 198
17.3 AGILITY EXAMPLE: RESET 198
17.3.1 Building and simulating the reset example 199
17.3.2 Simulating the reset example 199
17.3.3 Building the reset example for Hardware 199

18 TROUBLESHOOTING 200

19 ERRORS, WARNINGS AND INFORMATION MESSAGES 200

20 ERROR MESSAGES 200
20.1 ERRORS E1000 - E1999 201
20.1.1 Error code: E1002 203
20.1.2 Error code: E1003 203
20.1.3 Error Code: E1005 203
20.1.4 Error code: E1009 203
20.1.5 Error code: E1010 203
20.1.6 Error code: E1011 204
20.1.7 Error code: E1015 204
20.1.8 Error code: E1016 204
20.1.9 Error code: E1017 204
20.1.10 Error code: E1056 205
20.1.11 Error code: E1063 205
20.1.12 Error code: E1065 206
20.1.13 Error code: E1066 206
20.1.14 Error code: E1067 206
20.1.15 Error code: E1069 206
20.1.16 Error code: E1070 206
20.1.17 Error code: E1077 207
20.1.18 Error code: E1082 207
20.2 ERRORS: E2000 -E2999 210
20.2.1 Error code: E2017 212
20.2.2 Error code: E2081 212
20.2.3 Error code: E2082 212

21 WARNING MESSAGES 214
21.1 WARNINGS: W3000 - W3999 214

22 INFORMATION MESSAGES: I4000 - I4999 215

23 TROUBLESHOOTING: FIFOS 215

24 GLOSSARY 217
24.1 LEGAL NOTICES 220
24.2 AGILITY LICENSES 220

25 INDEX..... 221

Conventions

The following conventions are used in this document.

Warning Message. These messages warn you that actions may damage your hardware.

Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.

Sections of code or commands that you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

```
copy SourceFileName DestinationFileName
```

Optional elements are enclosed in square brackets like this:

```
struct [type_Name]
```

Curly brackets around an element show that it is optional but it may be repeated any number of times.

```
string ::= "{character}"
```

Synthesizable types are represented by ***T***.

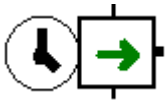
Graphics conventions

The following icons are used in the diagrams in this documentation.

A SystemC module

A SystemC process

SystemC port. The arrow indicates the data direction.



SystemC clock port

Connection to/from a port

Assumptions & Omissions

This manual assumes that you:

- have used SystemC or have the SystemC Language Reference Manual
- are familiar with common programming terms (e.g. functions)
- are familiar with your operating system (MS Windows or Linux)

This manual does not include:

- instruction in VHDL or Verilog
- instruction in the use of place and route tools

1 Celoxica Agility user manual

This user manual contains:

- a description of the SystemC subset that is synthesized by Agility
- a description of how the synthesis is performed
- a brief step-by-step guide to synthesizing a single-chip project
- instructions describing how to perform various useful tasks with Agility
- reference guides to the IDE and the command line compiler
- tutorials and examples
- some troubleshooting information

This guide assumes that you already know SystemC and can use standard MS Windows or Linux features.

Related topics

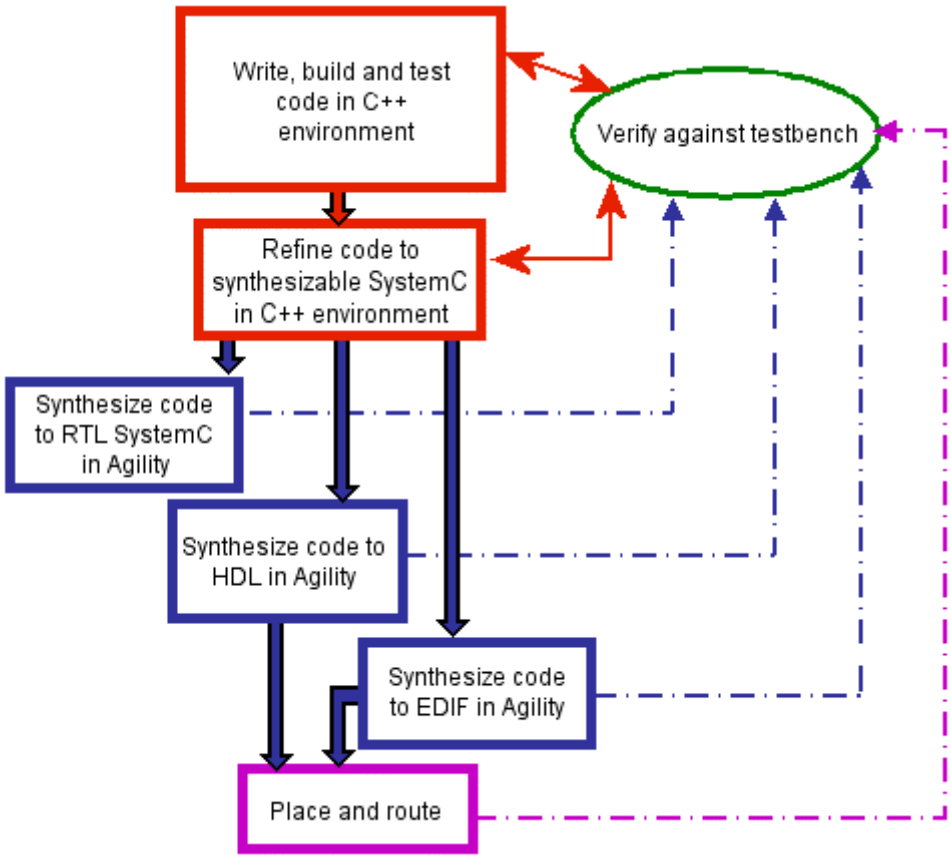
Agility IDE guide 154

1.1 Project development sequence

The normal development sequence is:

1. Write code and testbench files in a SystemC environment.
2. Build, simulate and test code in a SystemC environment.
3. Start Agility.
4. Create a new project.
5. Configure the project.
6. Add existing code files to the project.
7. Create source code file for synthesis start point and constraints.
8. Link to any required libraries.
9. Compile to RTL SystemC.
10. Simulate and test code using previous testbench in a SystemC environment.
11. Compile the project for the target chip.
12. Export the target file to a place and route tool.
13. Place and route. There is no information on placing and routing within the Agility documentation. Consult the documentation for your place and route tool.

1.1.1 Agility workflow


Write, build and test code in C++ environment
Verify against testbench
Refine code to synthesizable SystemC in C++ environment
Synthesize code to RTL SystemC in Agility
Synthesize code to HDL in Agility
Synthesize code to EDIF in Agility
Place and route

1.1.2 Agility walk-through

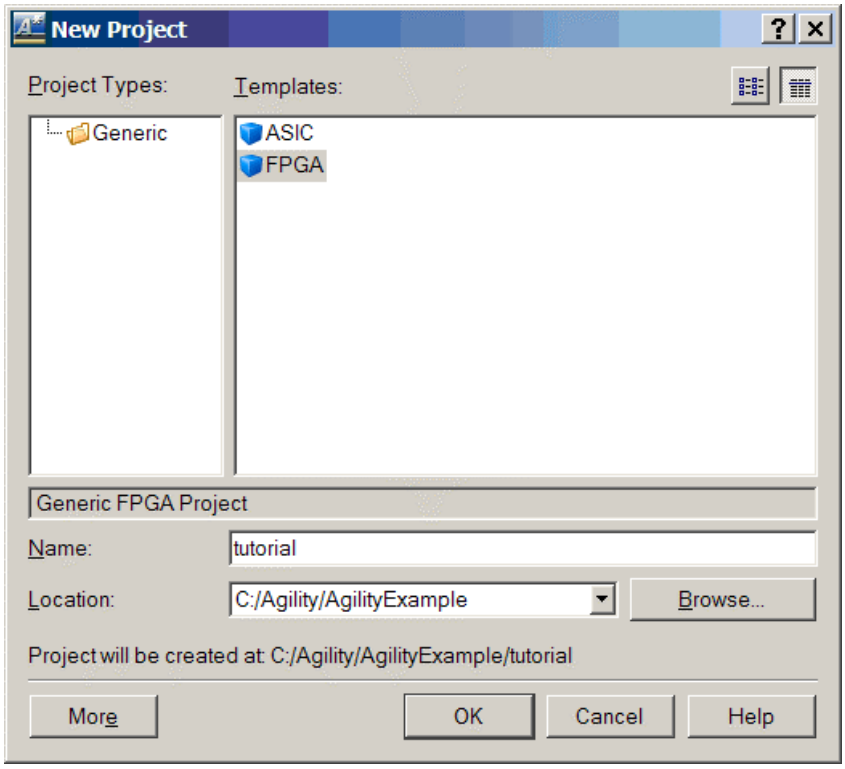
This short walk-through shows you how to:

- create an Agility project and workspace
- add existing files to the workspace
- view them in the code editor
- synthesize the project
- open the CDFG viewer.

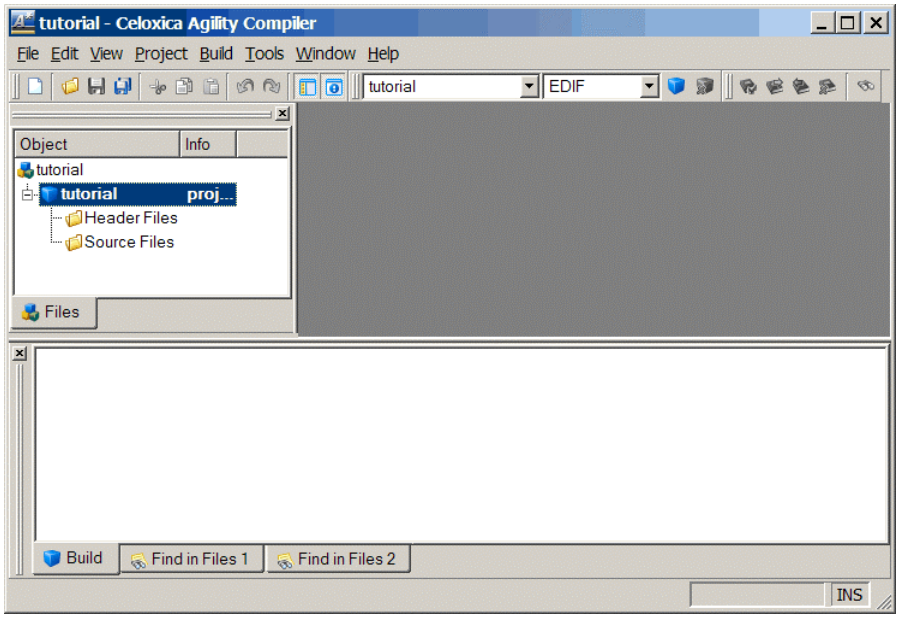
Creating a project in Agility

1. Start Agility by selecting Start>Program Files>Celoxica>Agility Compiler or double-clicking the Agility icon on the desktop.
2. Agility will open with a blank workspace window.
3. Select File>New>Project (or press Ctl Shift N) to open the New Project dialog.
4. Select FPGA as the type of project. This will use configuration settings appropriate to an FPGA rather than an ASIC
5. Type a project name (e.g. `tutorial`) in the Name: box.

6. Browse for the directory that you wish to create the project in.



7. Click OK. A new project will appear in the Agility workspace pane



By default, a workspace will be created in the same directory, with the filename ***Project***`.chw`.

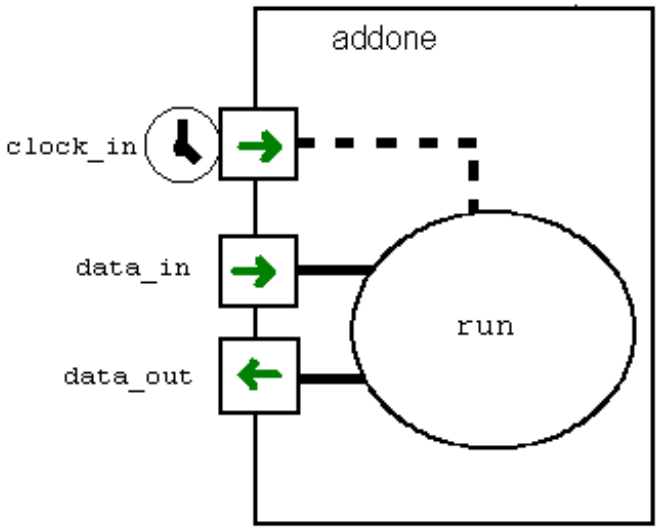
Creating a header file and adding it to the project

1. Right-click the Header Files icon in the workspace pane to add a new header file to the project.

   The header file will be used to hold the declarations of modules, helper functions and classes used in your design.
2. Name the file `tutorial.sc.h` and click Save. It will be saved to the project directory.
3. Double click the file icon in the workspace pane to open it in the code editor.
4. Add the following text to the file:

```
SC_MODULE(addone)
{
public:

    // Ports
    sc_in<bool> clock_in;
    sc_in<sc_int<16> > data_in;
    sc_out<sc_int<16> > data_out;


private:

    // Processes
    void run();


public:

   // Constructor
    SC_CTOR(addone)
    {
        SC_THREAD(run);
        sensitive << clock_in.pos();
    };
};
```

This header declares the module `addone`. It has a clock port, two data ports and a thread, `run`, which is sensitive to the positive edge of the clock.



Creating a source code file and adding it to the project

1. Right-click the Source Files icon in the workspace pane to add a new source file to the project.

   The source file will be used to hold the implementations of the modules, functions and classes declared in the header file.
2. Name the file `tutorial.sc.cpp` and click Save. It will be saved to the project directory.
3. Double-click the file icon in the workspace pane to open it in the code editor.
4. Add the following text to the file

```
#include <systemc.h>
#include "tutorial.sc.h"

void addone::run()
{
    // Reset cycle, before the first wait()

    data_out = 0;
    wait();

    while(1)
    {
        /*
        * Read data, add one and write it out every clock
        * cycle, forever
        */
        data_out.write(data_in.read() + (sc_uint<16>)1);
        wait();
    }
}
```

This defines the functionality of the `addone` member function `run`.

Creating a synthesis entry point file

1. Right-click the source file icon in the workspace pane to add a new empty source file to the project.

   This file is going to contain the `ag_main` function that is looked for by Agility as the start point of the design.
2. Call the file `synthesis.sc.cpp` and click Save. It will be saved to the project directory.
3. Double click the file icon in the workspace pane to open it in the code editor.
4. Add the following text to the file:

```
#include <systemc.h>
#include "tutorial.sc.h"
void ag_main()
{
    // An instance of addone is the top level module

    addone top_level("top_level");
}
```

This creates an instance of the `addone` module called `top_level` in the `ag_main` function.

The `ag_main` function is the synthesis entry-point. It must contain a single top-level module, which instantiates all other modules. (It is the equivalent of the `main` function in a C++ program.)

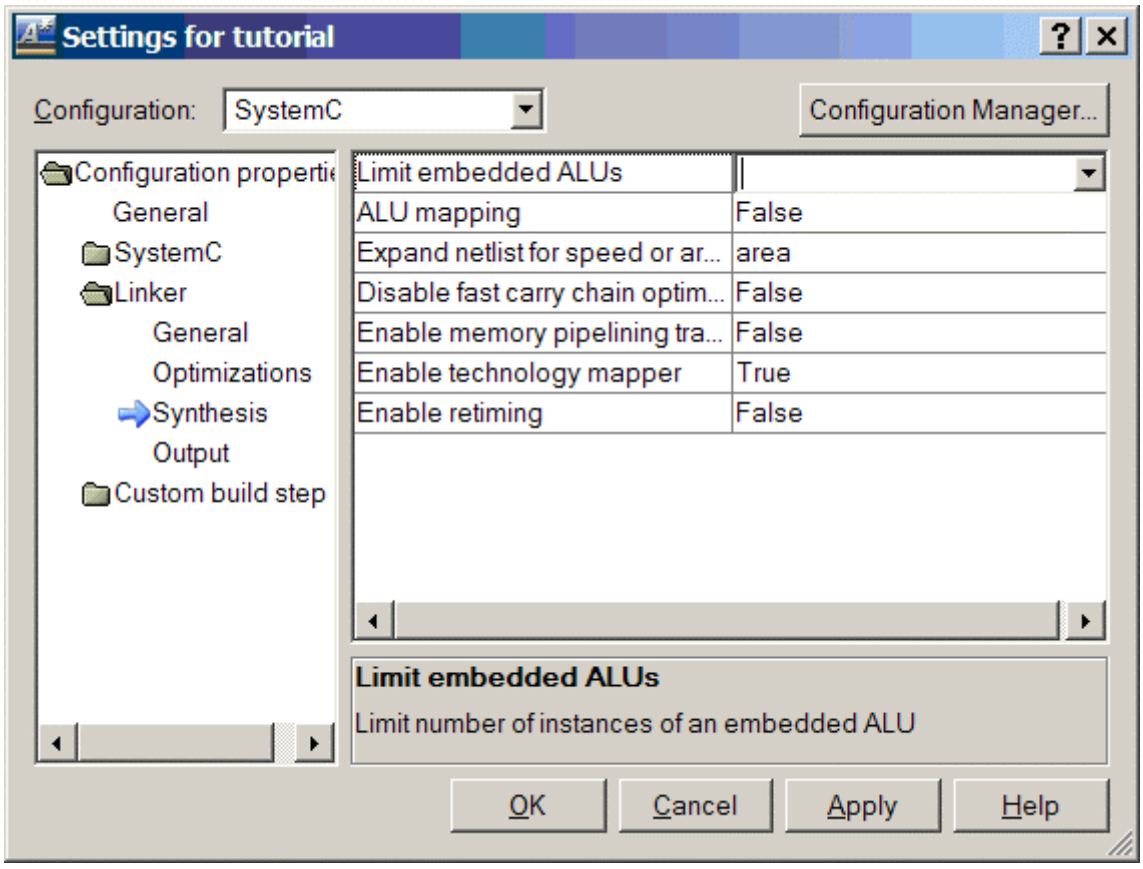
Simulation entry-point

The OSCI simulator uses an `sc_main` function as the simulation entry-point. If you wish to write a testbench for simulating your SystemC code, you would instantiate the top-level module within the `sc_main` function expected by the OSCI simulator, instead of the `ag_main` function expected by Agility

Setting the build configuration

Agility needs to know what synthesis output to produce. This is set for each project in the Project Settings.

1. Right-click the tutorial project name in the workspace and select Settings, or select the tutorial project and then Project>Settings.
2. This opens the Project Settings dialog. The default configuration is EDIF. This outputs EDIF which will be optimized to the selected part.
3. Select the Synthesis node in the Linker folder. Have a look at the different optimizations available.
4. Change the Configuration setting to SystemC. Note how the optimizations are switched off.



5. Click OK.

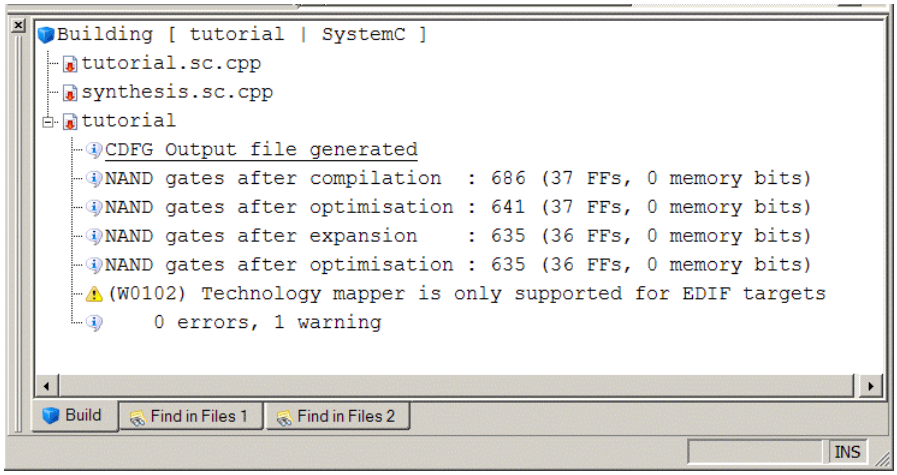
Building an RTL SystemC file and using the CDFG viewer

It is worth building with a SystemC output so that you can:

- quickly check that your SystemC design is synthesizable by Agility

- use the RTL SystemC output produced by Agility with an existing testbench.

1. Right-click the tutorial project and select Build, or select the tutorial project and press F7.
2. The project will start to build. A series of messages appear in the output window giving the project name and configuration type, and the files compiled within the project.


Building [tutorial | SystemC]
tutorial.sc.cpp
synthesis.sc.cpp
tutorial
CDFG Output file generated
NAND gates after compilation : 686 (37 FFs, 0 memory bits)
NAND gates after optimisation : 641 (37 FFs, 0 memory bits)
NAND gates after expansion : 635 (36 FFs, 0 memory bits)
NAND gates after optimisation : 635 (36 FFs, 0 memory bits)
(W0102) Technology mapper is only supported for EDIF targets
0 errors, 1 warning
Build
Find in Files 1
Find in Files 2
INS

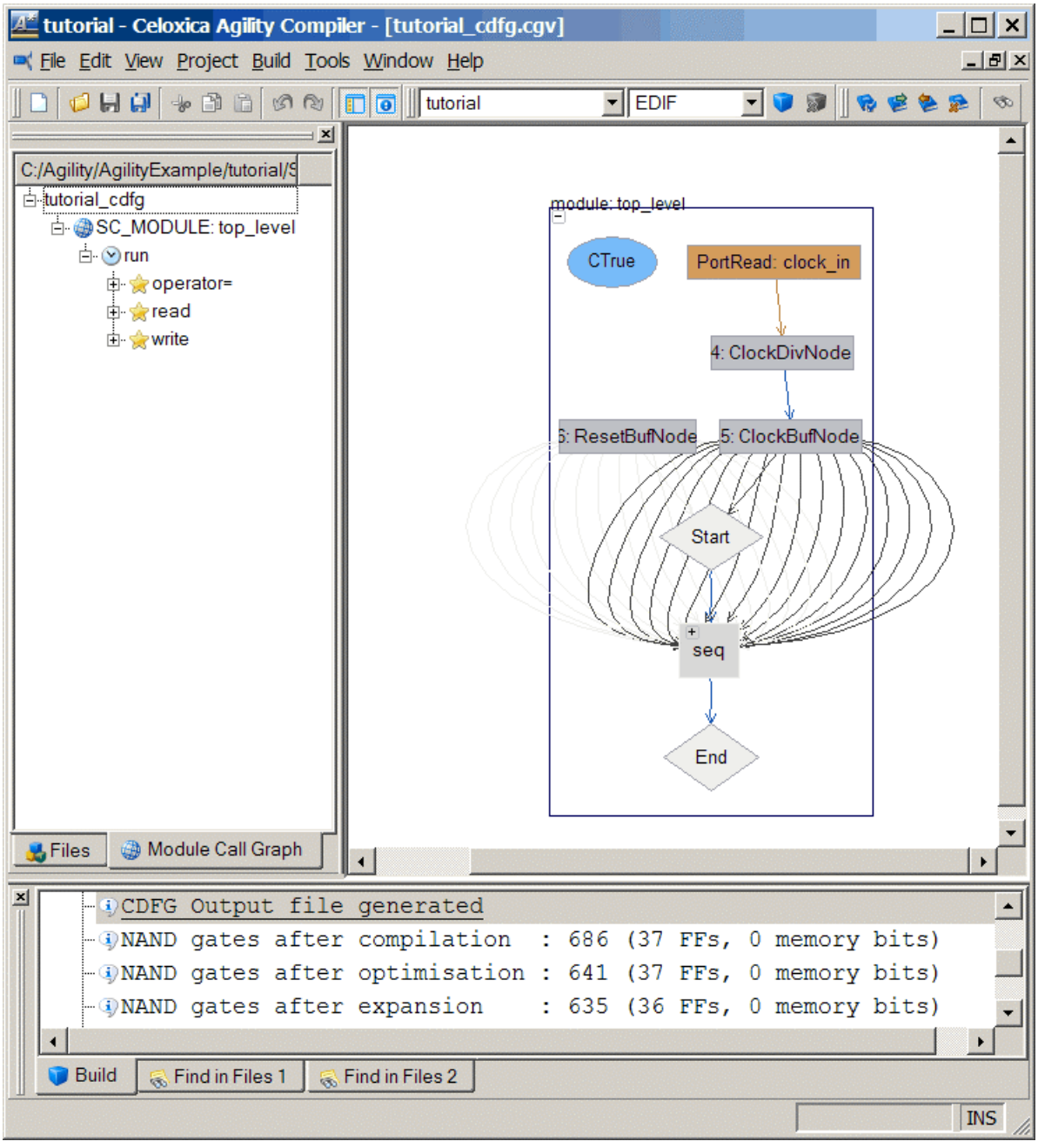
Viewing the CDFG file

The message `CDFG Output file generated` is a hyperlink. If you click it, the appropriate file will be loaded into the IDE.

1. Double-click `CDFG Output file generated`. The ***CDFG viewer*** (see page 166) opens.
2. The workspace pane switches to a Module Call Graph tab. This shows the call graph of the project.

   The right-hand pane (canvas) shows a grey rectangle representing the top-level module
3. You can double-click the rectangle to see what is inside the top-level module. You can also open the top-level module on the call graph to see what methods or threads are within it.

4. Select an item in the left pane and right-click to either draw that section of the graph or go to the source code it represents.



Building an EDIF netlist

To produce a file that can be placed and routed on an FPGA means that you have to specify the chip that you wish to use. This allows Agility to target the architecture of that chip.

1. Open the Project Settings dialog.(Right-click the tutorial project name in the workspace and select Settings)

2. Change the Configuration setting to `EDIF`.
3. Select the Output node in the Linker folder.
4. Select the target family (e.g. Xilinx Spartan-3/3L).
5. Enter a target part (e.g. xc3s1500fg320-4)
6. Right-click the tutorial project and select Build, or select the tutorial project and press F7.

The project starts to build. You will be given a summary of the gate and LUT usage:

```
NAND gates after compilation  : 686 (37 FFs, 0 memory bits)
NAND gates after optimisation : 641 (37 FFs, 0 memory bits)
NAND gates after expansion    : 603 (36 FFs, 0 memory bits)
NAND gates after optimisation : 476 (34 FFs, 0 memory bits)
LUTs after mapping            : 19 (34 FFs, 0 memory bits)
LUTs after retiming           : 19 (34 FFs, 0 memory bits)
    0 errors, 0 warnings
```

Viewing the Agility build reports

Two reports are produced by default when you build a project in Agility:

- the area and estimation summary.

  This summarises what Agility would expect your design to be like. It lists the longest paths etc. You may use this report to judge where to make changes to optimize your design.

- the XML report file.

  This describes the optimizations performed and includes any errors and warnings produced.

1. Select the File>Open command from the menu bar.
2. Browse to the output directory and open the file `tutorial_report.xml` in the IDE.
3. Scan it for information about your design.
4. Open the file `summary.html`.
5. See which of your files has the longest path.

Sample outputs

Sections of the following sample outputs from builds of the tutorial project are included to give a sense of the type of code produced by Agility.

- EDIF output
- VHDL output
- Verilog output
- RTL SystemC output

EDIF output

Here is a sample of the EDIF code generated by Agility:

```
(edif tutorial
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap
  (keywordLevel 0)
 )
 (status
  (written
   (timestamp 2005 10 21 14 41 1)
   (program "Celoxica Hardware Compiler"
    (version "Version 3.5.2842")
   )
  )
 )
 (library tutorial_lib
  (edifLevel 0)
  (technology
   (numberDefinition
   )
  )
  (cell FDSE
   (cellType GENERIC)
   (view view_1
    (viewType NETLIST)
    (interface
     (port C (direction INPUT))
     (port CE (direction INPUT))
     (port D (direction INPUT))
     (port S (direction INPUT))
     (port Q (direction OUTPUT))
    )
   )
  )
  (cell IPAD
   (cellType GENERIC)
   (view view_1
    (viewType NETLIST)
    (interface
     (port IPAD (direction OUTPUT))
     ..... <code snipped>.....
```

VHDL output

Here is a sample of the VHDL code generated by Agility:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
LIBRARY CELOXICA;
USE CELOXICA.CeloxicaPackage.ALL;
-- this is the top level entity
ENTITY tutorial IS
  -- interface description
  PORT (
    PADIN_communication_h_data_in_data_in_0 : IN std_logic;
    PADIN_communication_h_data_in_data_in_1 : IN std_logic;
    PADIN_communication_h_data_in_data_in_10 : IN std_logic;

    ......

    PADOUT_tutorial_sc_h_data_out_data_out_7 : OUT std_logic;
    PADOUT_tutorial_sc_h_data_out_data_out_8 : OUT std_logic;
    PADOUT_tutorial_sc_h_data_out_data_out_9 : OUT std_logic
  );
END;
```

```
-- this is the top level entity
ARCHITECTURE behaviour OF tutorial IS
  -- Power cable
  CONSTANT VCC : std_logic := '1';
  -- Ground cable
  CONSTANT GND : std_logic := '0';
  -- Clock cycle delay
  SIGNAL ControlWireId_4 : std_logic := '1';
  -- Clock cycle delay
  SIGNAL ControlWireId_56 : std_logic := '0';
  -- Bitwise OR
  SIGNAL ControlWireId_287 : unsigned(0 DOWNTO 0) := "0";
  -- Bitwise not
  SIGNAL ControlWireId_298 : unsigned(0 DOWNTO 0) := "0";
  -- Register 'Register_data_out'
  SIGNAL data_out : unsigned(15 DOWNTO 0) := (OTHERS => '0');
  -- Multiplexer
  SIGNAL DataWireId_291 : unsigned(15 DOWNTO 0) := (OTHERS => '0');
  -- Clock cycle delay
  SIGNAL ControlWireId_195 : std_logic := '0';
  -- Constant value
  CONSTANT ControlWireId_220 : unsigned(0 DOWNTO 0) := "0";
  -- Constant value
  CONSTANT ControlWireId_290 : unsigned(0 DOWNTO 0) := "1";
  -- Register 'Register_data_in'
  SIGNAL data_in : unsigned(15 DOWNTO 0) := (OTHERS => '0');
  -- Constant value
  CONSTANT DataWireId_147 : unsigned(15 DOWNTO 0) := "0000000000000001";
  -- Plus
  SIGNAL DataWireId_147_1 : unsigned(15 DOWNTO 0) := (OTHERS => '0');
  -- Concatenation of bits
  SIGNAL DataWireId_288 : unsigned(15 DOWNTO 0) := (OTHERS => '0');

BEGIN

-- Processes:

  PROCESS(PADIN_tutorial_sc_h_clock_in_clock_in_0) BEGIN
    IF rising_edge(PADIN_tutorial_sc_h_clock_in_clock_in_0) THEN
      ControlWireId_4 <= GND; -- Delay
      ControlWireId_56 <= ControlWireId_4; -- Delay
      ControlWireId_195 <= to_stdlogic(ControlWireId_287); -- Delay

      -- Update of register 'Register_data_in'
      IF (to_stdlogic(ControlWireId_290)='1') THEN
        data_in <= DataWireId_288;
      END IF;
```

```
      -- Update of register 'Register_data_out'
      IF (to_stdlogic(ControlWireId_287)='1') THEN
        data_out <= DataWireId_147_1;
      END IF;
    END IF;
  END PROCESS;


  PROCESS(ControlWireId_287, data_out, DataWireId_147_1) BEGIN
    -- Multiplexer
    IF (to_stdlogic(ControlWireId_287)='1') THEN
      DataWireId_291 <= DataWireId_147_1;
    ELSE
      DataWireId_291 <= data_out;
    END IF;
  END PROCESS;


-- Combinational expressions:

  DataWireId_288 <= to_unsigned(PADIN_communication_h_data_in_data_in_15) &
to_unsigned(PADIN_communication_h_data_in_data_in_14) &
    to_unsigned(PADIN_communication_h_data_in_data_in_13) &
to_unsigned(PADIN_communication_h_data_in_data_in_12) &
    to_unsigned(PADIN_communication_h_data_in_data_in_11) &
to_unsigned(PADIN_communication_h_data_in_data_in_10) &
    to_unsigned(PADIN_communication_h_data_in_data_in_9) &
to_unsigned(PADIN_communication_h_data_in_data_in_8) &
    to_unsigned(PADIN_communication_h_data_in_data_in_7) &
to_unsigned(PADIN_communication_h_data_in_data_in_6) &
    to_unsigned(PADIN_communication_h_data_in_data_in_5) &
to_unsigned(PADIN_communication_h_data_in_data_in_4) &
    to_unsigned(PADIN_communication_h_data_in_data_in_3) &
to_unsigned(PADIN_communication_h_data_in_data_in_2) &
    to_unsigned(PADIN_communication_h_data_in_data_in_1) &
to_unsigned(PADIN_communication_h_data_in_data_in_0);
  ControlWireId_287 <= ControlWireId_220 or to_unsigned(ControlWireId_56)
or to_unsigned(ControlWireId_195);
  DataWireId_147_1 <= data_in + DataWireId_147;
  ControlWireId_298 <= not(ControlWireId_287);
```

```
-- Port connections:

  PADOUT_tutorial_sc_h_data_out_data_out_0 <= to_stdlogic(DataWireId_291(0
DOWNTO 0));
  PADOUT_tutorial_sc_h_data_out_data_out_1 <= to_stdlogic(DataWireId_291(1
DOWNTO 1));
  PADOUT_tutorial_sc_h_data_out_data_out_2 <= to_stdlogic(DataWireId_291(2
DOWNTO 2));
  PADOUT_tutorial_sc_h_data_out_data_out_3 <= to_stdlogic(DataWireId_291(3
DOWNTO 3));
  PADOUT_tutorial_sc_h_data_out_data_out_4 <= to_stdlogic(DataWireId_291(4
DOWNTO 4));
  PADOUT_tutorial_sc_h_data_out_data_out_5 <= to_stdlogic(DataWireId_291(5
DOWNTO 5));
  PADOUT_tutorial_sc_h_data_out_data_out_6 <= to_stdlogic(DataWireId_291(6
DOWNTO 6));
  PADOUT_tutorial_sc_h_data_out_data_out_7 <= to_stdlogic(DataWireId_291(7
DOWNTO 7));
  PADOUT_tutorial_sc_h_data_out_data_out_8 <= to_stdlogic(DataWireId_291(8
DOWNTO 8));
  PADOUT_tutorial_sc_h_data_out_data_out_9 <= to_stdlogic(DataWireId_291(9
DOWNTO 9));
  PADOUT_tutorial_sc_h_data_out_data_out_10 <=
to_stdlogic(DataWireId_291(10 DOWNTO 10));
  PADOUT_tutorial_sc_h_data_out_data_out_11 <=
to_stdlogic(DataWireId_291(11 DOWNTO 11));
  PADOUT_tutorial_sc_h_data_out_data_out_12 <=
to_stdlogic(DataWireId_291(12 DOWNTO 12));
  PADOUT_tutorial_sc_h_data_out_data_out_13 <=
to_stdlogic(DataWireId_291(13 DOWNTO 13));
  PADOUT_tutorial_sc_h_data_out_data_out_14 <=
to_stdlogic(DataWireId_291(14 DOWNTO 14));
  PADOUT_tutorial_sc_h_data_out_data_out_15 <=
to_stdlogic(DataWireId_291(15 DOWNTO 15));

END;
```

Verilog output

Here is a sample of the Verilog code generated by Agility:

```
// this is the top level module
module tutorial ( clock_in, data_in, data_out );
  // interface description
  input clock_in;
  input [15:0] data_in;
  output [15:0] data_out;
```

```
// Power cable
wire VCC;
// Ground cable
wire GND;
// Not block
wire ControlWireId_4;
// Clock cycle delay
reg ControlWireId_56;
// Bitwise OR
wire [0:0] ControlWireId_287;
// Bitwise not
wire [0:0] ControlWireId_298;
// Register 'Register_data_out'
reg [15:0] data_out_1;
// Multiplexer
reg [15:0] DataWireId_291;
// Clock cycle delay
reg ControlWireId_195;
// Constant value
wire [0:0] ControlWireId_220;
// Constant value
wire [0:0] ControlWireId_290;
// Register 'Register_data_in'
reg [15:0] data_in_1;
// Constant value
wire [15:0] DataWireId_147;
// Plus
wire [15:0] DataWireId_147_1;
// Start block
reg NotStart;
```

```
  /* Always statements: */

  always @ (posedge clock_in)
    begin
      ControlWireId_56 <= ControlWireId_4; // Delay
      ControlWireId_195 <= ControlWireId_287; // Delay

      // Update of register 'Register_data_in'
      if (ControlWireId_290)
        data_in_1 <= data_in;

      // Update of register 'Register_data_out'
      if (ControlWireId_287)
        data_out_1 <= DataWireId_147_1;

      NotStart <= VCC; // Update of start block
    end


  always @ (ControlWireId_287 or data_out_1 or DataWireId_147_1)
    begin
      // Multiplexer
      if (ControlWireId_287)
        DataWireId_291 <= DataWireId_147_1;
      else
        DataWireId_291 <= data_out_1;
    end



  /* Combinational expressions: */

  assign ControlWireId_287 = ControlWireId_220 | ControlWireId_56 |
ControlWireId_195;
  assign DataWireId_147 = 16'h1;
  assign DataWireId_147_1 = data_in_1 + DataWireId_147;
  assign ControlWireId_290 = 1'b1;
  assign ControlWireId_220 = 0;
  assign ControlWireId_298 = ~(ControlWireId_287);
  assign ControlWireId_4 = ~(NotStart);
  assign VCC = 1'b1;
  assign GND = 0;


  /* Port connections: */

  assign data_out = DataWireId_291;

endmodule
```

RTL SystemC output

Here is a sample of the RTL code generated by Agility:

```
#include "tutorial.h"

const sc_uint<1> tutorial::VCC = 0x1;
const sc_uint<1> tutorial::GND = 0;
const sc_uint<1> tutorial::ControlWireId_220 = 0;
const sc_uint<1> tutorial::ControlWireId_290 = 0x1;
const sc_uint<16> tutorial::DataWireId_147 = 0x1;


void tutorial::Interface0()
{
  data_out = sc_int<16>(DataWireId_291.read());
}

void tutorial::Sequential0()
{
  static bool Startup = true;

  if (Startup)
  {
    Startup = false;
    ControlWireId_4 = 0x1;
    data_in_1 = 0;
    data_out_1 = 0;
    ControlWireId_56 = 0;
    ControlWireId_195 = 0;
  }

  else
  {
    ControlWireId_4 = GND; // Delay

    // Update of register 'Register_data_in'
    if (ControlWireId_290)
      data_in_1 = sc_uint<16>(data_in.read());

    // Update of register 'Register_data_out'
    if (ControlWireId_287.read())
      data_out_1 = DataWireId_147_1.read();

    ControlWireId_56 = ControlWireId_4.read(); // Delay
    ControlWireId_195 = ControlWireId_287.read(); // Delay
  }

}
```

2 Celoxica SystemC synthesizable subset

This document describes the SystemC language features supported by Agility Compiler. This document does not attempt to teach SystemC or C++.

For further information see:

ISO/IEC 14882 International Standard - Programming Languages - C++.
American National Standards Institute, New York (1998).

Draft Standard SystemC Language Reference Manual 2.1.
Open SystemC Initiative (2005).
(Available from ***http://www.systemc.org*** (http://www.systemc.org))

SystemC Synthesizable Subset Draft
(currently under review by the OSCI Synthesis Working Group):

The C++ Programming Language
Bjarne Stroustrup
(Third Edition)
Addison-Wesley
ISBN 0-201-88954-4

The C Programming Language
B. W. Kerninghan and D. M. Ritchie.
Second Edition.
Prentice Hall, New Jersey (1988).

2.1 Agility SystemC synthesizable subset overview

SystemC is a C++ class library that allows you to:

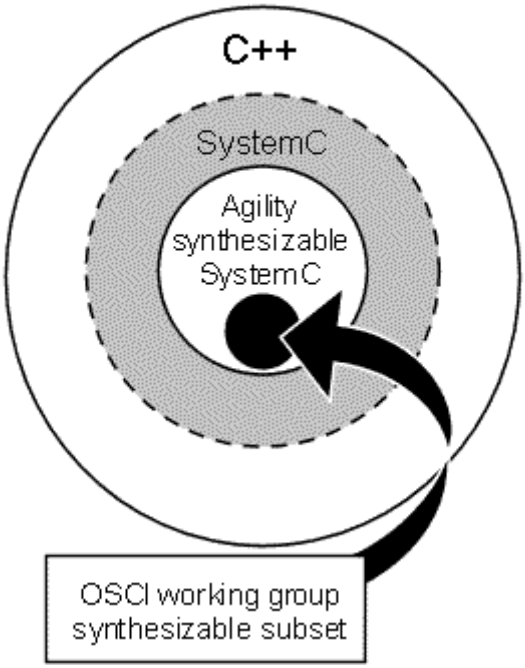
- model
- develop
- test
- synthesize

  a system in a single language, with the same testbench throughout.

Agility is a tool to synthesize SystemC. Non-synthesizable parts must be placed in software instead of hardware, or rewritten in a synthesizable form.

Agility supports a wider synthesizable subset than that proposed by OSCI. Some of these extensions enable you to synthesize certain hardware features such as RAMs.

Extensions that are specific to Agility are prefixed by `ag_` or `AG_`. Standard SystemC keywords are prefixed by `sc_` or `SC_`.



2.1.1 The synthesizable subset

Synthesizable code is code that can:

- be converted into gates and storage elements (or wires and registers).
- perform operations and store data of fixed size.

Synthesizable code cannot:

- have run-time dynamic operations, such as memory allocation or deallocation.

The standard synthesizable subset of SystemC is defined in a working document by OSCI (the Open SystemC Initiative). The OSCI website is ***www.systemc.org*** (http://www.systemc.org/).

Agility also synthesizes some SystemC constructs which are not included in the OSCI synthesizable subset.

2.2 Modules

Modules are the basic building blocks in SystemC. They can be defined by:

- using the `SC_MODULE` macro
- inheriting from the SystemC class `sc_module`.

These are both defined in `systemc.h`.

Member functions and data can be declared inside a module and defined outside the module in the usual way.

Templatized modules
Modules may be templatized to allow parameterization at the point of use. This is supported for synthesis. For example:

```
template< unsigned N >SC_MODULE( AndGate )
{
    sc_in< sc_uint< N > > Inputs;
    sc_out< bool > Output;
    SC_CTOR( AndGate ) { }
};
```

declares a module that has one output port and a single input port of width `N`. The actual value for `N` is specified when the module is instantiated:

```
AndGate<3> MyAndGate;
```

You can specify a default value for `N` in the template. In the code below, `N` will be 2 unless a different value is specified:

```
template< unsigned N = 2 >SC_MODULE( AndGate )
{
    // Definitions
};
```

2.2.1 The SC_MODULE macro

An `SC_MODULE` declaration is of the form

```
SC_MODULE(module_name)
{
   ... // port declarations to connect modules together (see page 40)

   ... // variable declarations (see page 52)

   ... // function declarations/definitions

   SC_CTOR(module_name)
   {
    ... // body of constructor
    ... // process declarations, sensitivities
   }
};
```

The arguments to `SC_MODULE` and `SC_CTOR` must be the same.

SC_CTOR

The `SC_CTOR` constructor is used to:

- Initialize variables declared in the module.

- Specify the functionality of the module in terms of SC_(C)THREADS and SC_METHODS. The threads and methods must be defined using synthesizable code.

The constructor should also contain:

- The sensitivity lists describing the inputs that each process is sensitive to.
- Instantiation of sub-modules.
- Port mapping code for hierarchical modules.

Example

```
SC_MODULE( half_adder )
{
    sc_in<bool> a,b;
    sc_out<bool> sum, carry;

    void half_adder_defn();

    SC_CTOR( half_adder )
    {
        SC_METHOD ( half_adder_defn );
        sensitive << a << b;
    }
};


void half_adder::half_adder_defn()
{
    sum = a ^ b;
    carry a & b;
}
```

Writing your own constructor

You may define your own module constructor. If your module contains sub-processes, you will need to invoke the SC_HAS_PROCESS macro:

```
SC_HAS_PROCESS(MyModule);
MyModule(sc_module_name name_, int InitialValue_) :
           sc_module(name_), InitialValue(InitialValue_)
{
   ... // synthesizable SystemC code
}
```

In this example, InitialValue is a member variable of myModule.

Both SC_CTOR and SC_HAS_PROCESS are supported for synthesis.

Use either `SC_CTOR` or `SC_HAS_PROCESS` in a module, not both.

Inheriting from the sc_module class

An alternative way of defining a module is to declare a new class that is derived from the base class `sc_module`:

```
struct myModule : public sc_module
{
   ... // port declarations

   ... // member variables

   ... // function declarations/definitions

   SC_HAS_PROCESS(myModule);
   myModule( sc_module_name module_name ) : sc_module( module_name )
   {
      ... //  functionality of module
   }
};
```

Modules may be defined using classes and `struct`s, and these may inherit multiple base classes/`struct`s.

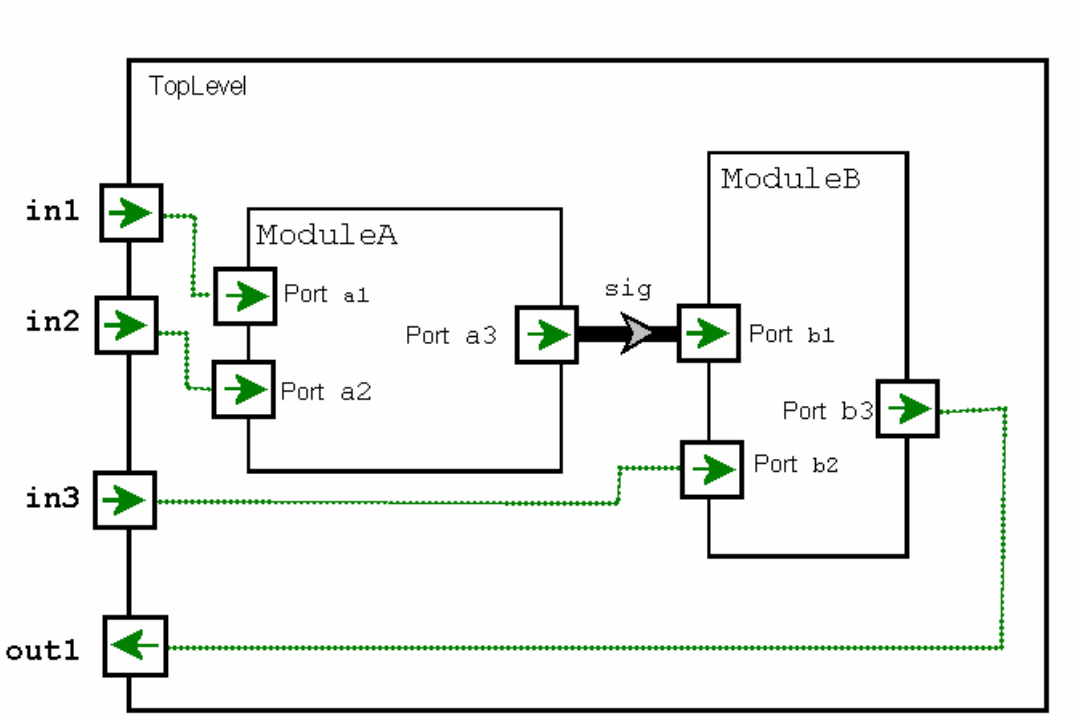
Virtual inheritance is not supported for synthesis.

2.2.2 Hierarchical modules

A module may contain sub-module instances in its body to form hierarchy.

There are no restrictions on the level of nesting of the hierarchy. Top-level modules can have hierarchy.

For example:



The `TopLevel` module contains two sub-modules, called `ModuleA` and `ModuleB`, communicating with one another via a signal (`sig`) and to the outside world via ports. In SystemC, you could declare the top-level module by writing:

```
SC_MODULE(TopLevel)
{
    // port declarations
    sc_in<sc_uint<8> > in1;
    sc_in<sc_uint<8> > in2;
    sc_in<sc_uint<8> > in3;
    sc_out<sc_uint<8> > out;

    // module instances
    ModuleA *myModuleA;
    ModuleB *myModuleB;

    // signal declarations
    signal<sc_uint<8> > sig;

    SC_CTOR(TopLevel)
    {
        ... // port mapping code
        ... // creating module instances
    }
};
```

where `ModuleA` and `ModuleB` are module declarations.

Creating sub-module instances

Sub-module instances may be constructed directly as member data, or by using the `new` operator.

Construction of member data

```
ModuleA *myModuleA;
ModuleB *myModuleB;

...

SC_CTOR(MyTopModule):
    myModuleA("myModuleA"), myModuleB("myModuleB")
    {
        ...
    }
```

Construction using the new operator

```
ModuleA *myModuleA;
ModuleB *myModuleB;

...

SC_CTOR(MyTopModule)
{
    myModuleA = new ModuleA("myModuleA");
    myModuleB = new ModuleB("myModuleB");
...
}
```

2.2.3 Port types

Ports are the means through which modules communicate with other modules. There are three basic port types:

- input ports for receiving data
- output ports for sending out data
- input/output ports which combine the two.

Input ports

An input port must be of type `sc_in`, which is a templatized primitive SystemC class. SystemC provides `typedef`s for common specializations of `sc_in`:

- `sc_in_clk` – supported for synthesis
- `sc_in_resolved` – not supported for synthesis
- `sc_in_rv` – not supported for synthesis

Usage
`sc_in< T > PortName;`

declares an input port of type ***T***.

`sc_in_clk clkName;`

declares an input port of type `bool`.

Example
```
sc_in< sc_int<8> > InputPort;
```

Output ports

An output port must be of type `sc_out`, which is a templatized primitive SystemC class. SystemC provides `typedef`s for common specializations of `sc_out`:

- `sc_out_clk` – supported for synthesis
- `sc_out_resolved` – not supported for synthesis
- `sc_out_rv` – not supported for synthesis

By default, output ports are initialized to zero. To leave them undefined, use the `AG_UNDEFINED` (see page 75) macro

Usage
`sc_out< T > PortName;`
declares an output port of type ***T.***

`sc_out_clk clkName;`
declares an output port of type `bool`.

Example
`sc_out< sc_uint<5> > OutputPort;`

Input/Output ports

An inout port must be of type `sc_inout`, which is a templatized primitive SystemC class. SystemC provides `typedef`s for common specializations of `sc_inout`:

- `sc_inout_clk` – supported for synthesis
- `sc_inout_rv` (tristate port) – supported for synthesis
- `sc_inout_resolved` – supported for synthesis

Usage
`sc_inout< T > PortName;`
declares a port for both input and output of data of type ***T***.

`sc_inout_clk clkName;`

declares an inout port of type `bool`.

Example
`sc_inout< sc_uint<5> > IOPort`;

Tristate I/O ports

Tristate ports can be declared for bi-directional communication over buses.

Usage

`sc_inout_rv<`***width***`>` ***PortName***`;`
declares a tristate inout port of width ***width***.

`sc_inout_resolved` ***PortName***`;`
declares a tristate inout port of width 1.

To prepare the port for reading, a `Z` must be written to each bit of the port. (A string of `Z`'s of length ***n*** to `sc_inout_rv`, or a single char '`Z`' to `sc_inout_resolved`.)

Example

```
SC_MODULE(RamAccess)
{
    sc_in<bool>          clock;
    sc_out<bool>         write_enable;
    sc_out<sc_uint<16> > address;
    sc_inout_rv<16>      data;

    void run()
    {
        wait();
        sc_uint<16> count, value;

        while (1)
        {
            count ++;
            data.write("ZZZZZZZZZZZZZZZZ");
            write_enable.write(false);
            address.write(count);

            wait();

            value = data.read();
            if (value & 1)
                value *= 3;
            else
                value /= 2;
            data.write(value);
            write_enable.write(true);

            wait();
        }

    }
```

```
    SC_CTOR(RamAccess)
    {
        SC_THREAD(run);
        sensitive << clock.pos();
    }
};
```

The example continually cycles through the memory locations halving even values and trebling odd values.

User-defined ports

`sc_port` is a primitive templatized class provided in `systemc.h`.

You may define your own port types directly by inheriting from `sc_port`, and specializing it with a particular interface.

Synthesizable operations associated with ports

Two operations are associated with ports: `read` and `write`.

They are both in the synthesizable subset.

- `sc_inout` ports can be written to and read from;
- `sc_in` ports can only be read from.
- `sc_out` ports can only be written to.

Usage

```
sc_in< T > PortName;
T TheData = PortName.read();

sc_out< T > PortName;
PortName.write( data );

sc_inout< T > PortName;
PortName.write( data );
T TheData PortName.read();
```

Port mapping

There are two ways to map ports in SystemC:

- named - supported for synthesis
- positional - not supported for synthesis.

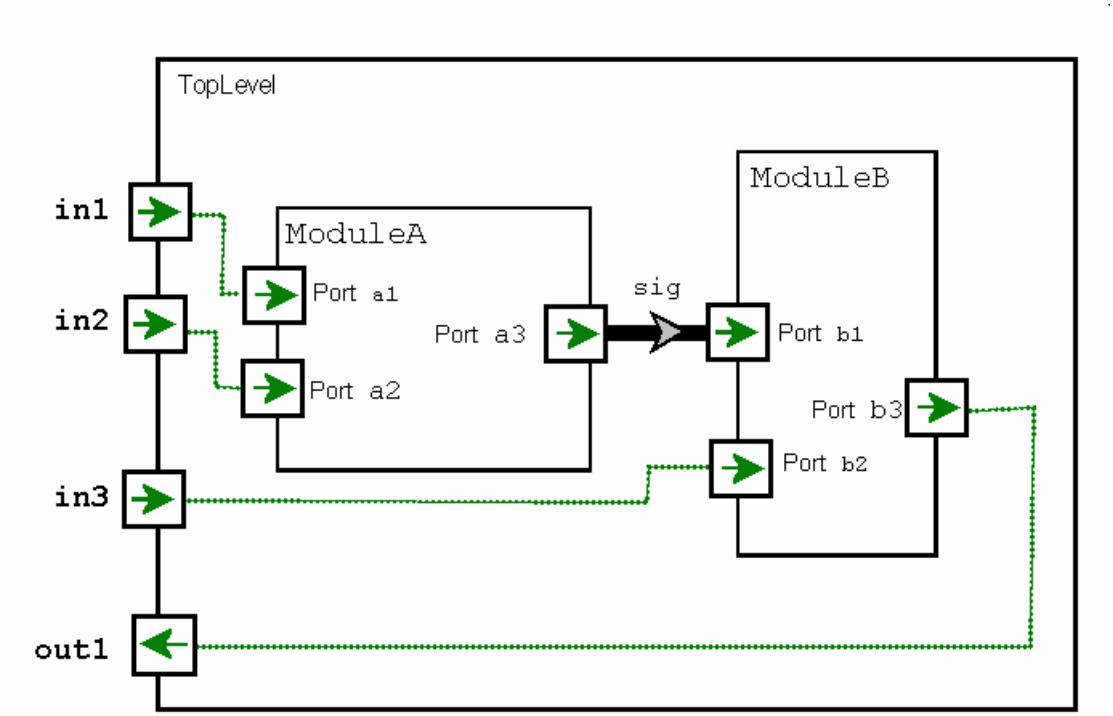
Example: Named mapping

Suppose that there is a top-level module, containing two submodules, A and B. The following ports are to be mapped to external ports (shown by the dotted lines):

- `in1` to `a1`

- `in2` to `a2`
- `in3` to `b2`
- `b3` to `out1`

Ports `a3` and `b1` are to be connected by the signal `sig`.



Example

```
SC_MODULE(TopModule)
{
    sc_in<sc_uint<8> > in1;
    sc_in<sc_uint<8> > in2;
    sc_in<sc_uint<8> > in3;
    sc_out<sc_uint<8> > out;

    ModuleA *myModuleA;
    ModuleB *myModuleB;

    signal<sc_uint<8> > sig;
    SC_CTOR(TopLevelModule)
    {
        myModuleA = new ModuleA("ModuleA");
        myModuleB = new ModuleB("ModuleB");


        myModuleA->a1(in1);
        myModuleA->a2(in2);
        myModuleA->a3(sig);

        myModuleB->b1(sig);
        myModuleB->b2(in3);
        myModuleB->b3(out);

    };
```

Ports can be mapped in any order.

2.2.4 Channels

Channels are used for communication and synchronization. Agility supports two types of channel:

- signals
- FIFOs.

Signals

Signals are used to interconnect threads and methods. The `sc_signal` class is a primitive SystemC class that takes a template argument specifying the type of the signal. The latter has to be a primitive type (i.e. cannot be another channel).

The following special signals are not supported in Agility:

- `sc_signal_resolved`
- `sc_signal_rv`

By default, signals are initialized to zero. To leave them undefined, use `AG_UNDEFINED` (see page 75).

Signals can only be assigned one value in between two events.

Different signal assignment statements located in different processes must not write to the same signal or port.

Usage

`sc_signal< T > SignalName;`

declares a signal of type *T*.

Example

```
sc_in< sc_int<8> > MySignal;
```

Initialization

Signal data is initialized to zero for integer types. To construct a signal with un-initialized data, you must use the Agility extension `AG_UNDEFINED`.

FIFOs

FIFOs are used to transfer data between processes through a buffer. The `sc_fifo` class is a primitive SystemC class that takes a template argument specifying the type of the FIFO data. `sc_fifo` has member functions for blocking (`read`, `write`) and non-blocking (`nb_read`, `nb_write`) access, which are all supported by Agility.

You can specify the synchronization timing of FIFOs in a clock domain by using the ***FIFO constraints*** (see page 93).

Usage

`sc_fifo< T > FIFOName;`

declares an FIFO with data of type *T*.

Example

```
sc_fifo< sc_int<8> > MyFIFO;
```

Restrictions

The member functions `num_available()` and `num_free()` are not supported by Agility.

2.2.5 Processes

Agility supports all three types of SystemC process:

- SC_THREAD
- SC_CTHREAD

  (This is a special case of SC_THREAD)
- SC_METHOD

Processes exist within modules. Each process consists of two parts:

- Process declaration
- Process body

Process declaration

The process declaration must exist within its parent module's constructor. The process's sensitivity to clocks, resets and other signals and ports are specified when the process is declared.

Example

```
my_module.h:
SC_MODULE (my_module)
{
   void my_thread();
   sc_port <bool> clock;
   ...
   SC_CTOR (my_module)
   {
      SC_THREAD (my_thread);
      sensitive << clock.pos();
   }
  ...
};
```

Process body

The process body contains the implementation of the process.

It may be defined:

- within the module definition, typically in a .h header file (for example, my_module.h)
- outside the module, typically in a .cpp file.

Example 1

A thread process body within a module definition.

```
SC_MODULE (my_module)
{
   void my_thread()
   {
      ...
   }
   ...
};
```

Example 2
A thread process body outside a module definition.

```
SC_MODULE (my_module)
{
   void my_thread();
   ...
};

my_module.cpp:
void my_module::my_thread()
{
   ...
}
```

Thread processes

A thread process can be used to synthesize synchronous hardware. There are two types of thread: `SC_THREAD` and `SC_CTHREAD`.

`SC_THREAD`s have an initialization and synthesis stage. The initialization stage is performed before the rest of the process is executed. It is performed when the circuit is started.

`SC_CTHREAD`s do not have an initialization stage.

Reset

Both types of thread can be reset with a global asynchronous reset.

- `SC_THREAD`s perform the initialization stage on reset
- `SC_CTHREAD`s abandon the current execution and restart from the beginning on reset.

`SC_CTHREAD`s may also be reset locally using a synchronous reset.

Suspend and resume

Both types of thread have a clock and active edge associated with them. The thread suspends upon the use of a `wait()` statement and resumes upon an event from the thread's associated clock edge.

Thread processes can only be sensitive to clocks and resets.

`return` is not permitted in a thread process.

wait statements in thread processes

`wait()` statements delimit the clock edges in a thread process.

Any statements between two `wait()` statements will be constructed as combinational logic. These two examples therefore create the same logic:

```
// Extremely compact code
wait();
c = ((a&0xF0)>>4)|((b&0x0F)<<4);
wait();

// More comprehensible code
wait();
c = (a&0xF0);
c = c >> 4;
d = (b&0x0F);
d = d << 4;
e = c | d;
wait();
```

`wait(n)` and `wait_until()` are not supported by Agility. `wait_until()` can be implemented using `do.. while`.

SC_THREAD processes

An `SC_THREAD` process body consists of two stages separated by a `wait()` statement:

- compile-time initialization
- run-time synthesized hardware.

The only control flow between the initialization stage and synthesized stage must be through the `wait()` statement separating them.

Example

```
void my_thread()
{
    ... // compile-time initialization stage

    wait();
    ... // run-time hardware synthesis stage
}
```

The synthesis stage runs when it receives the signal to which the process is sensitive. The thread may be made sensitive to a positive edge or a negative edge but not both.

```
SC_THREAD (my_thread);
        sensitive << clock.pos();
```

Initialization stage

The initialization stage ends at the first `wait` statement.

All the values assigned to variables in the initialization stage must be resolvable at compile-time. Therefore they cannot contain signal or port reads

Variables assigned to during initialization and read in the synthesized stage are synthesized as hardware constants.

Variables assigned to in both the initialization stage and the synthesis stage are synthesized as registers with reset. The value used for the reset will be the value of variable at the end of the initialization stage.

Further compile-time evaluation will take place where possible within the synthesis stage.

Synthesis stage

The synthesis stage of a thread process should typically be written as a non-terminating loop. If this synthesis stage terminates, the corresponding synthesized hardware enters a locked state. You can only exit this state by using a global reset.

Example

```
void my_module::run() // SC_THREAD
{
    int a, b;

    // end of compile-time initialization stage
    wait();     // Wait for first clock edge
    // start of run-time synthesized hardware stage

    a = 1;      // Clock cycle 1
    wait();
    a = a + 1;  // Clock cycle 2
    b = 5;      // Clock cycle 2
    wait()
    a = b;      // Clock cycle 3
    b = b + 1;  // Clock cycle 3
    wait();
    ...
}
```

SC_CTHREAD processes

A clocked thread process differs from a thread process in the way it is declared and how it responds to a reset.

- The clock is specified as an argument to the `SC_CTHREAD` macro.
- No sensitivity list is used with `SC_CTHREAD`s.
- If a reset occurs and the process has not terminated, the process will be restarted.

  Reset signals or ports are declared by calling `reset_signal_is` after declaring the `SC_CTHREAD`.

  The second argument to `reset_signal_is` specifies whether the reset signal is active high or active low.

Example

```
SC_MODULE (my_module)
{
    sc_in <bool> clock;
    sc_in <bool> reset;

    void my_cthread ();

    SC_CTOR (my_module)
    {
        SC_CTHREAD (my_cthread, clock.pos())
        reset_signal_is (reset, true);
    }
};
```

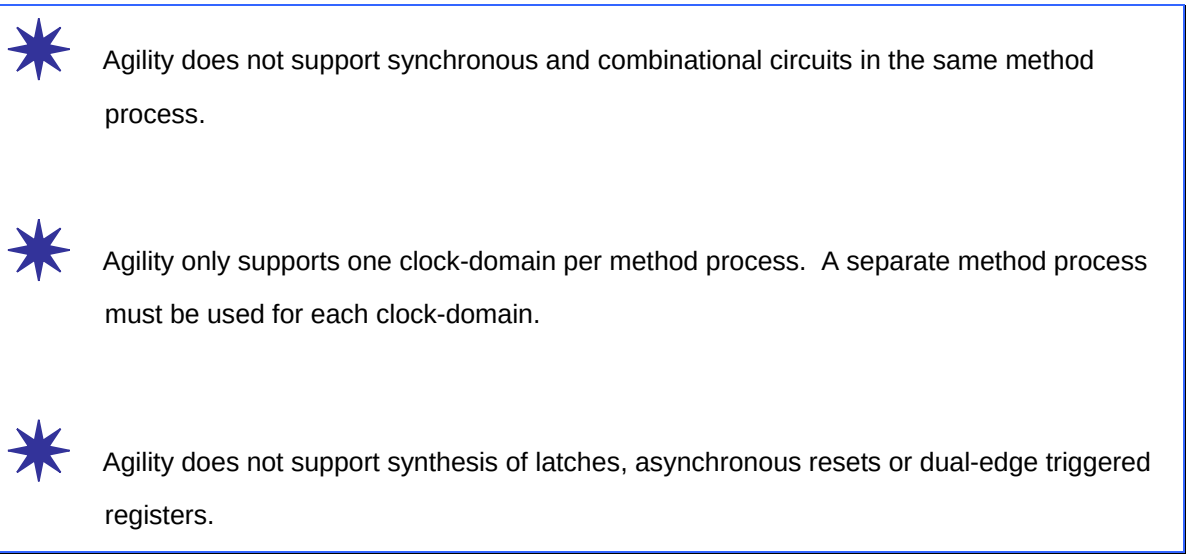
SC_METHOD processes

`SC_METHOD` processes differ from thread processes. They

- must not contain `wait()` statements;
- must always terminate.

A method process can be used to model either synchronous hardware or combinational hardware.

Process restrictions


Agility does not support synchronous and combinational circuits in the same method process.
Agility only supports one clock-domain per method process. A separate method process must be used for each clock-domain.
Agility does not support synthesis of latches, asynchronous resets or dual-edge triggered registers.

Combinational circuits

To synthesize a combinational circuit:

- the method must be sensitive to all the ports and signals that it reads.
- each signal and port written to must be written to on every execution of the `SC_METHOD` .

- a variable may only be read from if it has previously been written to within the same execution of the `SC_METHOD`.

These requirements restrict the use of variables such that they can only be used to hold values for use within a single execution of the method.

Example: Combinational adder

```
SC_MODULE (Adder)
{
    sc_in <sc_uint<32> > in1;
    sc_in <sc_uint<32> > in2;
    sc_out <sc_uint<32> > out;

public:
    void add()
    {
        out = in1.read() + in2.read();
    }

    SC_CTOR (Adder)
    {
        SC_METHOD (add);
        sensitive << in1 << in2;
    }
};
```

Synchronous circuits

Method processes can also be used for synchronous circuits. You can specify a clock by:

- testing for a clock edge within the body of the method process and sensitizing the process to that clock,

  OR

- sensitizing the method process to one edge of one clock and nothing else.

Agility supports `posedge()` and `negedge()` for testing rising and falling clock edges respectively.

Agility supports `pos()` and `neg()` for sensitizing a process to just one edge of a clock signal.

Example: Synchronous adder

```
SC_MODULE (Adder)
{
    sc_in <bool> clock;
    sc_in <sc_uint<32> > in1;
    sc_in <sc_uint<32> > in2;
    sc_out <sc_uint<32> > out;

public:
    void add()
    {
        if (clock.posedge())
        {
            out = in1.read() + in2.read();
        }
    }

    SC_CTOR (Adder)
    {
        SC_METHOD (add);
        sensitive << clock;
    }
};
```

Spawning threads

Agility supports spawning threads within `SC_FORK`/`SC_JOIN` pairs. The threads spawned will be dynamic in simulation and static in synthesis. The syntax is:

```
SC_FORK
    sc_spawn(parameters);
     ...
    sc_spawn(parameters);
SC_JOIN
```

sc_spawn

The supported form of `sc_spawn` is:

```
template <typename T>
sc_process_handle sc_spawn(
    T myObject,
    const char * spawnedProcessName = 0,
    const sc_spawn_options* myOptions = 0);
```

where

myObject is a pointer to the function to be spawned or the function object

spawnedProcessName is a unique name to be given to the spawned process

myOptions is a pointer to an object specifying the process sensitivities

In order to be synthesizable, the process can only be sensitive to a clock. Hence, the only `sc_spawn_options` methods supported are:

```
template<typename T>
void set_sensitivity( const sc_in<T> &port );

template<typename T>
void set_sensitivity( const sc_signal<T> &sig );

void set_sensitivity( const sc_event *the_event );
```

For further information on spawning threads, consult the ***SystemC Language Reference manual*** http://www.systemc.org.

sc_spawn restrictions

Agility has the following restrictions upon `sc_spawn`.

- Spawned processes must have must be sensitive to the same clock as the calling thread
- Users may not spawn `sc_method`s.
- The options passed to the spawned processes must be declared in the initialization stage of a thread.
- Users may not spawn processes that have a return type other than `void`.
- `sc_spawn` may only be used within `SC_FORK`/`SC_JOIN` pairs.
- `SC_FORK`/`SC_JOIN` pairs may only be called from the body of a process.

Example

```
void my_thread()
{
    sc_spawn_options opts;
    opts.set_sensitivity( clock.pos() );
    wait();
    SC_FORK
        sc_spawn( fn_object1(), "fn_object1", &opts ),
        sc_spawn( fn_object2(), "fn_object2", &opts )
    SC_JOIN
}
```

Spawned process return type workaround

Functions with return types other than `void` can be refactored so that the values are passed by reference.

Workaround

```
sc_int<16> foo( )
{
    return 4;
}
```

can be altered to:

```
void foo( sc_int<16> & return_arg )
{
    return_arg = 4;
}
```

sc_spawn options restrictions

Agility does not support these member functions of `sc_spawn_option`s.

- `dont_initialize`
- `set_stack_size(`***N***`)`
- `spawn_method`

2.3 Datatypes

This section describes Agility datatypes supported for synthesis.

Agility supports some data types for compile-time evaluation but not for synthesis.

Some data types are only supported for synthesis if their values are compile-time determinable.

The list of supported datatypes is available in the ***summary*** (see page 65).

2.3.1 C++ datatypes

Agility supports all C++ datatypes for synthesis, unless otherwise specified. There are some restrictions relating to whether values are compile-time determinable.

C++ integer types

Declaration

`long long` ***Name;***

`long` ***Name***;

`int` ***Name;***

`short` ***Name;***

`char` ***Name;***

`bool` ***Name;***

Header
None

Description
Agility supports C++ integer types for compile-time evaluation and synthesis.

Datatype	***Width (bits)***
`signed long long` `unsigned long long`	64
`signed` `unsigned long`	32
`signed` `unsigned int`	32
`signed short int` `unsigned short int`	16
`signed char` `unsigned char`	8
`bool`	1

Examples
```
// Assigned from a decimal literal.
int a = 10;
// Assigning from a character constant.
signed char c = 'h';

// Assigning a hex literal.
unsigned long b = 0xFFFFFFFFUL;

// Assigning a 64 bit value.
unsigned long long v = 0xFF00FF00CC00CC00ULL;
```

C++ floating-point types

Declaration
`long double` ***Name;***

`double` ***Name;***

`float` ***Name;***

Header
None

Description
Agility supports the following primitive C++ floating-point types for compile-time evaluation only. Any calculation involving floating point must evaluate to a constant. Agility reports an error if they do not.

Datatype	*Smallest*	*Largest*
`float`	`1E-38`	`1E+38`
`double`	`1E-308`	`1E+308`
`long double`	`1E-308`	`1E+308`

Examples

```
sc_int<16> SineTable[128];
for ( int i = 0; i < 128; i++ )
{
   double index = (double) i;
   double angle = 2.0 * PI * (index / 128.0);
   double sineangle = sin( angle );
   SineTable [i] =  sc_uint<16>( sineangle *  32767.0);
}
```

This shows the use of floating points to set up a sine table. You could use the `ag_constrain_ram` constraint to place the array in a RAM.

```
float f = 2.71e-4;
double p = 3.142d;
long double l = 2.232e10l;

sc_in<float> in;
float x;
x = in.read();
```

The `float` cannot be synthesized as its value cannot be determined at compile-time.

```
sc_out<int> out;
float x = 5.54;
out.write(x);
```

The floating-point value is implicitly cast to the type of the port when it is written. (The value written will be 5.)

Operations on C++ types behave as they would in a C++ software compiler.

Operations on C++ types differ from those applying to SystemC types of equal width.

E.g., `int` is NOT equivalent to `sc_int<32>`

For more information consult a C++ book or the C++ standard.

Array types

Declaration

***T Name*`[`*size*`];`**

Header
None

Description
Arrays are supported in Agility. An array is synthesizable if its elements are of a synthesizable type and its size is compile-time determinable.

Synthesizable examples

```
int temp, array[100];
sc_in<int> in;
for (int y=0; y<100; y++)
{
    in.read(temp);
    array[y] = temp;
    wait();
}
```

The `int` type is synthesizable so an array of `int`s can be synthesized.

```
#include <math.h>
//...
float l[10];
for (int y=0; y<10; y++)
    l[y] = log10((y+1)*10);
//...
sc_out<int> out;
for (int y=0; y<10; y++)
    out.write(l[y]);
```

Although the array is not synthesizable, the members are used only during compile-time evaluation and are cast to type `int`.

Pointers and references

Agility supports both pointers and references, subject to the restriction that Agility can always determine the target of the pointer or reference.

Pointer types

Declaration
T * Name;

Header
None

Description
A pointer is synthesizable if it is a pointer to a synthesizable type, and the value to which the pointer points is compile-time determinable.

Pointers are permitted to be NULL.

Restrictions
Pointer arithmetic is not supported for either compile-time evaluation or synthesis.

Agility does not support pointers to class member variables or functions.

Example: Resolvable pointer
```
void clear(char * a, char * b)
{
   *a = 255;
   *b = 255;
}
// ...
sc_out<unsigned char> out;
unsigned char x, y;
init(&x, &y);
out.write(x);
```

The value written to the port would be 255.

Example: Non-resolvable pointer
```
if ( InPort.read() > 100 )
{
    p = &a;
}
else
{
    p = &b;
}
OutPort.write( *p );
wait();
```

The target of `p` depends on the value read in from `InPort` at runtime, and so cannot be determined. Agility will report an error.

Reference types

Declaration
T `&` ***Name;***

Header
None

Description
Agility supports reference types. A reference type is synthesizable if the reference target type is synthesizable and the reference target is compile-time determinable.

In many cases references will be compile-time determinable because references, unlike pointers, cannot change their target once assigned or reference NULL.

Agility does not support pointers whose target cannot be determined at compile-time. Since in C++ a reference is equivalent to a pointer (with the restriction that it must always point to one object for its whole lifetime) you also cannot conditionally assign a value to a reference if the condition cannot be determined at compile-time.

Example: Non-determinable reference

```
int b = 4, c = 5;
int &a = InPort.read() ? b : c;

// Not allowed, as we do not know until run-time which of 'b'
// or 'c' will be pointed to by 'a'
OutPort.write( a );
```

Classes

Declaration

```
struct X
{
// members.
};

class Y
{
// members.
};
```

Header

None.

Description

Agility supports class types. They can be synthesized if all members that are used have synthesizable types. (This permits you to use non-synthesizable member types and functions in simulation.)

Virtual functions are supported, with the restriction that the type of the object on which the function is called must be compile-time determinable.

Examples

```
struct Point
{
    int x;
    int y;
    float getfloatx() { return (float)x / (float)100 };
    float getfloaty() { return (float)x / (float)100 };
};
Point p;
sc_int<int> px, py;
p.x = px.read();
p.y = py.read();
```

The methods of `Point` that return `float` are not used here, so the code can be synthesized. The non-synthesizable methods may be required elsewhere for simulation.

```
struct Point
{
    int n;
    float x, y;
};
Point p0;
sc_in<Point> in;
p0 = in.read();
```

This cannot be synthesized. `Point` is not a synthesizable type because it contains `float`s that are not synthesizable.

Virtual inheritance

Agility does not support virtual inheritance.

Workaround

If a common base class contains no data members (for instance, the common base class specifies an interface) you can remove the `virtual` keyword and still get the same behaviour.

If the common base class does contain data members, you will need to modify the code to allow Agility to compile it.

Example

In the following example, classes `Derived1` and `Derived2` could inherit non-virtually from `BaseInterface` and no behaviour would change.

```
class BaseInterface
{
    virtual int GetNum();
};

class Derived1 : public virtual BaseInterface
{
    ...
};

class Derived2 : public virtual BaseInterface
{
    ...
};

class MultiDerived : public Derived1, public Derived2
{
    ...
};
```

Union types

Agility does not support union types for synthesis.

Workaround

You can use casts to convert between types in circumstances where a union type might be used. (These conversions could be encapsulated in a class if necessary.)

Bit fields

Agility does not support C bit fields for synthesis.

Workaround

You can get equivalent functionality by using structures containing SystemC types with widths such as `sc_uint` and `sc_biguint`.

2.3.2 SystemC numeric types

Agility supports SystemC integer and fixed-point types.

Supported integer types are:

- `sc_bigint`
- `sc_biguint`
- `sc_int`
- `sc_uint`

Supported ***fixed-point types*** (see page 62) are:

- `sc_fixed`
- `sc_ufixed`

SystemC integer types

Declaration

`sc_int<`***width***`>` ***Name;***

`sc_uint<`***width***`>` ***Name;***

`sc_bigint<`***width***`>` ***Name;***

`sc_biguint<`***width***`>` ***Name;***

By default, SystemC integer types are initialized to zero. To leave them undefined, use the `AG_UNDEFINED` ***macro*** (see page 75)

Header

```
#include <systemc.h>
```

Description

Agility supports the following SystemC integer data types:

Datatype	***Maximum Width***
`sc_int<`***width***`>`	Up to 64 bits
`sc_uint<`***width***`>`	Up to 64 bits
`sc_bigint<`***width***`>`	Unrestricted
`sc_biguint<`***width***`>`	Unrestricted

Agility supports the arithmetic, logical, bitwise and comparison operators on integers.

In addition, Agility supports the `iszero`, `sign`, `bit`, `range`, `reverse`, `test`, `set`, `clear`, `invert` and `length` functions.

For details of these operations consult OSCI ***SystemC Language Reference Manual*** http://www.systemc.org

Restrictions

- A left-shift operation must be followed immediately by a cast or assignment to a type of known width.
- A `range` operation must be followed immediately by a cast or assignment to a type of known width.

`range` operations are not allowed on intermediate results of `sc_biguint` or `sc_bigint` expressions.

The parameters to a `range` operation must be compile-time-determinable.

Examples: Valid shift

```
sc_int<32> a, b, c;
// ...
a = b << c;
```

An assignment immediately follows the left-shift.

```
sc_int<32> a, b, c, d;
// ...
a = (b << c) + d;
```

The result of the left-shift is not immediately cast or assigned.

```
sc_int<32> a, b, c, d;
// ...
a = ((sc_int<32>)(b << c)) + d;
```

The result of the left-shift is cast to a type of known width.

Example: Invalid shift

```
sc_int<32> a, b, c, d;
// ...
a = (b << c) + d;
```

The result of the left-shift is not immediately cast or assigned.

Examples: Valid range

```
sc_int<32> a, b, c = 1, d = 5;
a = b.range(c, d);
```

The expression is immediately assigned to a type of known width.

```
sc_int<32> a, b, c = 1, d = 5;
a = ((sc_int<4>)(b.range(c, d))) + 4;
```

The expression is cast to a type of known width before the addition operation.

Examples: Invalid range

```
sc_in<sc_int<32> > in;
sc_int<32> a=12, b, c;
in.read(c);
b = a.range(c,1);
```

The value of c cannot be determined at compile-time.

```
sc_int<32> a, b, c = 1, d = 5;
a = b.range(c, d) + 4;
```

The expression is not immediately cast or assigned.

SystemC fixed-point types

Declaration

`sc_fixed<`***Width, IntWidth, QuantMode, OverflowMode, SatBits***`> Name;`
`sc_ufixed<`***Width, IntWidth, QuantMode, OverflowMode, SatBits***`> Name;`

Parameter	***Meaning***	***Default***
Width	Width of the variable in bits	None
IntWidth	The number of bits in the integer part	None
QuantMode	Quantization mode	`SC_TRN` (truncation)
OverflowMode	Overflow mode used	`SC_WRAP` (wraparound)
SatBits	Number of saturated bits used for wrap	0

Header

```
#include <systemc.h>
```

Description

Agility supports the `sc_fixed` and `sc_ufixed` fixed-point data types for compile-time evaluation and synthesis.

Agility supports bitwise, arithmetic and comparison operations on `sc_fixed` and `sc_ufixed` types.

All the various quantization and overflow modes are available and behave as the OSCI simulation environment and as described in the ***SystemC reference manual*** http://www.systemc.org.

Restrictions

Types such as `sc_fix`, `sc_ufix` and `sc_fxnum` are not supported by Agility.

Conversion to and from floating-point values is supported only where values can be determined at compile-time.

Agility does not support the fixed-point context, so if the quantization and overflow mode are not set when a type is declared the same default (`SC_TRN` and `SC_WRAP` respectively) will always be used.

Literals

The following literals are used to define quantization mode.

Quantization Mode	*Parameter*
Truncation (default)	SC_TRN
Rounding to +∞	SC_RND
Truncation to 0	SC_TRN_ZERO
Rounding to 0	SC_RND_ZERO
Rounding to -∞	SC_RND_MIN_INF
Rounding to ∞	SC_RND_INF
Convergent rounding	SC_RND_CONV

The following literals are used to define overflow mode.

Overflow mode	*Parameter*
Wrap-around basic (default)	SC_WRAP, n_bits = 0
Saturation	SC_SAT
Symmetrical saturation	SC_SAT_SYM
Saturation to zero	SC_SAT_ZERO
Wrap-around advanced	SC_WRAP, n_bits > 0
Sign magnitude wrap-around	SC_WRAP_SM, n_bits ≥0

Examples

```
sc_fixed<8, 4> x = 0.5;   // 0000.1000
sc_ufixed<8, 4> y = 2.25;  // 0010.0100
sc_fixed<8, 4> z = x + y; // 0010.1100
```

The fixed-point numbers are created from floating-point values. No quantization or saturation has taken place.

```
double p = 2.5625;
sc_fixed<8,4> q = p;        // q = 0010.1001
sc_fixed<6,4> r = q;        // r = 0010.10
double s = r.to_double();  // s = 2.25
```

Quantization takes place using the default mode (SC_TRN). The floating-point conversions are all done at compile-time.

```
double p = 2.625;
sc_fixed<8,4,SC_RND,SC_WRAP> q = p;        // q = 0010.1010
sc_fixed<6,4,SC_RND,SC_WRAP> r = q;        // r = 0010.11 (= 2.75)
```

Quantization takes place using rounding to positive infinity (SC_RND).

2.3.3 SystemC bit vector and logic vector types

Declaration

`sc_bit` ***Name;***
`sc_bv<`***width***`>` ***Name;***
`sc_logic` ***Name;***
`sc_lv<`***width***`>` ***Name;***

Header

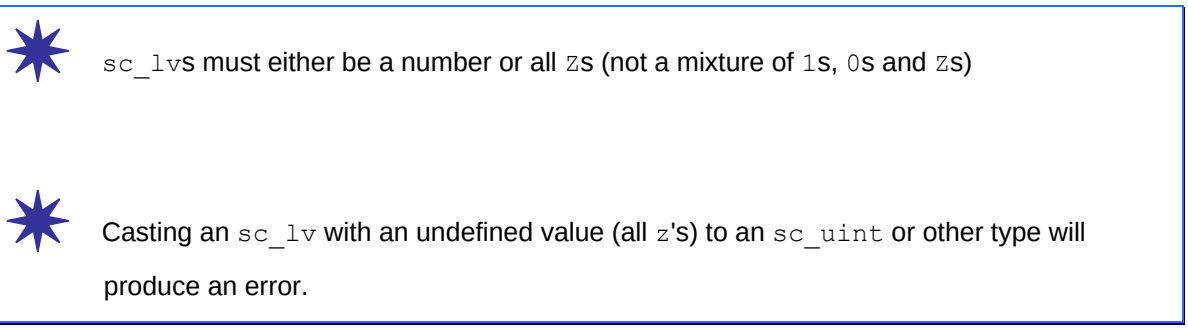
```
#include <systemc.h>
```

Description

Agility supports bit vector and logic vector types for both synthesis and compile-time evaluation. These are arrays of single bit values that do not have arithmetic operators defined. The following bit vector and logic vector types are available:

Datatype	***Width***
`sc_bit`	Always 1 bit
`sc_bv<`***width***`>`	Unrestricted
`sc_logic`	Always 1 bit
`sc_lv<`***width***`>`	Unrestricted

By default, `sc_bv` instances are initialized to 0. To leave them undefined, use the `AG_UNDEFINED` (see page 75) macro.



`sc_lv`s must either be a number or all `z`s (not a mixture of `1`s, `0`s and `z`s)

Casting an `sc_lv` with an undefined value (all `z`'s) to an `sc_uint` or other type will produce an error.

2.3.4 Summary

Type	*Available for compile-time evaluation*	*Available for synthesis*

`char`
`short int`
`int`
`long int`
`long long int`
	√	√

`float`
`double`
`long double`
	√	x

`bool`	√	√
Pointers	√	If compile-time resolvable
References	√	If compile-time resolvable
Unions	x	x
Bit fields	x	x
Arrays	√	Depends on contents
Structs	√	Depends on contents
Classes	√	Depends on contents

`sc_int`
`sc_uint`
`sc_bigint`
`sc_biguint`
	√	√

`sc_bit`
`sc_bv`
`sc_logic`
`sc_lv`
	√	√

`sc_fixed` `sc_ufixed`	√	√

Non-constant global/shared variables are not supported for synthesis.

2.4 Type promotion and conversions

Type promotion rules for C/C++ are not always intuitive and are dependent on the implementation being used. Agility follows standard typing rules for C++ for expressions containing C++ types. Users in any doubt are advised to consult a ***C++ reference book or the standard itself*** (see page 31).

For SystemC types, Agility implements the type conversion rules described in the SystemC Language Reference Manual at ***http://www.systemc.org*** http://www.systemc.org. This is consistent with the OSCI simulator.

This generally means that when C++ types are used together with SystemC types, the C++ types will be promoted to SystemC types. (This does not apply when a C++ type is shifted by a SystemC type.)

There are differences between C++ and SystemC type conversions which may cause confusion:

- Operands of the same type
- Minus operations

Operands of the same type
In SystemC the result of an operation involving two operands of the same type grows to accommodate the maximum size of the result, even if this is a different type.

C++ types will not grow like this, but will wrap at the specified number of bits.

There are two exceptions:

- Operands of a C++ type smaller than signed `int` are promoted to `signed int` before the operation is carried out. So an operation involving two chars or shorts will result in a `signed int`, rather than wrap.
- The result of an operation involving SystemC `sc_int` or `sc_uint` types will wrap at 64 bits. The `sc_bigint` and `sc_biguint` types have no such limits.

Minus operations
In SystemC the result of a minus or unary minus operation is always signed, even if the operands are unsigned.

In C++ if the operands to minus were unsigned the result would be unsigned. However there is an exception to this rule:

- If operands of a C++ type are smaller than `signed int` they are promoted to `signed int` before the operation is carried out. So a minus operation on two `unsigned char`s or `unsigned short`s results in a `signed int` result.

Example
```
// Assign 2^32-1 to 32 bit C++ types.
unsigned int x = 4294967295, y = 4294967295;
sc_uint<64> z = x + y;

// Assign the same values to equivalent SystemC types.
sc_uint<32> p = 4294967295, q = 4294967295;
sc_uint<64> r = p + q;
```

The example will compile and synthesize, but `z` is not equal to `r`. The expression involving C++ wrapped at 32 whereas the one involving SystemC types grew in size.

2.5 Variables

If a module body contains a variable declaration, the variable must be accessed exclusively by one process of the same module. If this were allowed, non-determinism could arise.

Suppose that `test_module` is a SystemC module containing a member variable `a` and executing two `SC_THREAD`s `test1()` and `test2()` in parallel, where `test1()` and `test2()` are defined thus:

```
void test_module::test1()
{
    a = 0;       // variable initialized to 0
    int x;
    wait();
    while( 1 )
    {
        x = a;   // reading variable a
        Result = x;
        wait();
    }
}

void test_module::test2()
{
    wait();
    while( 1 )
    {
        a++;    // reading from variable a and writing to variable a
        Result = 0;
        wait();
    }
}
```

Variable `a` is read in `test1()`, and both read from and written to in `test2()`. The two readings, although in the same clock cycle, may get different values.

2.6 Expressions and statements

C++ semantics is followed, except for the restrictions given.

Restrictions
`range` operator

is not supported on temporary variables from `sc_bigint` and `sc_biguint` operations. For example:

```
sc_biguint<5> a, b, r;
r = (a+b).range(5,0);
```

is not synthesizable by Agility

2.6.1 Expressions

Operators

Agility implements the operations on integers that are defined by the SystemC standard. These include the usual comparison operators, arithmetic operators, logic and bitwise logic operators.

In addition to standard operators the following SystemC operations are synthesizable using member-function syntax on SystemC integer types: `iszero`, `sign`, `bit`, `range`, `reverse`, `test`, `set`, `clear`, `invert` and `length`.

For details of these operations consult the OSCI ***SystemC Language Reference Manual*** http://www.systemc.org

Restrictions

- A left-shift operation must be followed immediately by a cast or assignment to a type of known width.
- A `range` operation must be followed immediately by a cast or assignment to a type of known width.
- The arguments to a `range` operation must be compile-time determinable.
- Agility does not support the use of the ternary operaotr on the LHS. Hence you cannot use the construct below:

```
int walk_the_dog = false, go_to_work = false;
```

```
(day == WEEKEND ? walk_the_dog : go_to_work) = true;
```

Operator new

Operator `new` is supported at compile time, but not at run-time. As the return type from `new` is a pointer, Agility pointer restrictions apply when using the allocated memory.

Operator delete

Operator `delete` is not supported.

2.6.2 Statements

Conditional statements (`if`, `switch`, loops etc) are only synthesizable if the expression and all of the statements within their associated block(s) are synthesizable.

if statements

There are no further restrictions on `if` statements.

switch statements

A `switch` is synthesizable if the expression, labels and statements enclosed within the associated actions are all synthesizable.

Restrictions

Each action in a switch must have a `break` statement. Fall through is not allowed. For example, the following is not synthesizable:

```
...
switch( option )
{
    case 0:
    {
        y = y + 1;
    }

    case 1:
    {
        z = z + 2;
        break;
    }

    default:
    {
        z = 0;
        break;
    }
}
```

as there is no break in the action corresponding to label `0`.

To make the above synthesizable, with the same behaviour, the fall-through code would have to be added to the action for label `0`:

```
...
switch( option )
{
    case 0:
    {
        y = y + 1;
        z = z + 2;
        break;
    }

    case 1:
    {
        z = z + 2;
        break;
    }

    default:
    {
        z = 0;
        break;
    }
}
```

Loops

The Agility synthesizable subset contains three types of loop:

- `for`
- `while`
- `do-while`

The associated condition, counters and block statements must all be synthesizable.

There are no other restrictions on loops.

2.7 Functions

If a function is to be synthesized, its body must only contain code that is within the Agility ***synthesizable subset*** (see page 52).

Restrictions

- Function parameters must be of a ***synthesizable type*** (see page 52).
- Arguments to functions must be compile-time determinable.

- General recursion is not supported for synthesis. For example, the following cannot be synthesized:

```
int sum(int x)
{
    if (x == 0 )
    {
       return 0;
    }
    else
       return ( 1 + sum(x-1) );
}

...
int x = sigIn.read();
SigOut.write( sum (x) );
```

Recursive definitions that can be ***resolved at compile-time*** (see page 118) are synthesizable.

2.8 Namespaces

A namespace allows you to define a new scope, avoiding name collisions of variables, types, classes or functions. You will find these useful when defining libraries. Agility supports the following for synthesis:

- anonymous namespaces
- named namespaces
- nested namespaces.

Namespace aliasing and the `using` keyword are also supported.

Namespace members that are to be synthesized must be defined using code within the Agility synthesizable subset.

2.9 Exceptions

Exceptions are not supported by Agility.

2.10 Transaction-level modelling

Transaction level modelling allows you to specify methods used to communicate, while keeping the implementation of those methods private.

2.11 Agility extensions

Agility supports the following extensions to the OSCI SystemC synthesizable subset.

- `ag_assert_`***assertionType***
  Compile-time asserts on a specified conditions
- `ag_foreign_module`
  Talk to a sub-component with no threads. You define the ports, but not the functionality.
- `ag_global_async_reset_is`
  Define a reset for the whole design
- `ag_undefined`
  Initialize a variable to an undefined value
- `ag_signal_default`.
  Set a default value on a signal.
- `sc_fifo`

The following functionality is supplied for compile-time use:

- `float`s and `double`s (these are only to be used for compile-time constants rather than run-time variables)
- compile-time determinable virtual functions
- `math.h` library (also only used to provide compile time constants)

2.11.1 Agility-specific types and functions

All Agility-specific types and functions are prefixed with `ag_` or `AG_` to distinguish them from standard SystemC types and functions. They are available in the header file `agility.h`.

2.11.2 Black boxes

Modules for which the implementation is not defined in Agility are referred to as “black boxes”. In order to instantiate a black box, the user must declare a black box class that inherits from the Agility-specific class `ag_foreign_module`. Instances of this class behave like instances of `sc_module` and thus can be instantiated in other modules. The constructors of black box classes may contain ***port constraints*** (see page 114) to set the port types in VHDL. However, black box classes cannot contain threads or methods.

The type of the black box component to be instantiated and its ports are taken from the name of the black box class and its ports respectively. An instance name may be passed to the constructor of `ag_foreign_module` to be able to identify the corresponding instance after synthesis.

Example: Black box

```
#include "systemc.h"
#include "agility.h"

/*
 * Declare a black box of type 'mult'
 */
class mult : public ag_foreign_module
{
public:

  /*
   * Black box ports
   */
  sc_in<sc_uint<32> > A, B;
  sc_out<sc_uint<64> > O;

  SC_CTOR(mult) : ag_foreign_module(name)
  {
  }
};

SC_MODULE(parent)
{
public:

  sc_in<bool> clock_in;
  sc_out<sc_uint<64> > result;

  /*
   * Black box declaration
   */
  mult *my_mult;

  sc_signal<sc_uint<32> > a, b;
  sc_signal<sc_uint<64> > o;

private:

  void run();
public:

  /*
   * Constructor
   */
  SC_CTOR(parent)
  {
   /*
```

```
    * Create an instance of mult with instance name 'my_mult'
    */
   my_mult = new mult("my_mult");

   SC_THREAD(run);
   sensitive << clock_in.pos();

    /*
     * Map the black box ports
     */
    my_mult->A(a);
    my_mult->B(b);
    my_mult->O(o);
  };
};
```

In the example above, the component to be instantiated is of type `mult` and its instance is called `my_mult`.

2.11.3 Global asynchronous reset

Agility provides a function called `ag_global_async_reset_is`, which defines a global asynchronous reset for the whole design. This is defined as a member function of `sc_module` for synthesis. If a global asynchronous reset is required, this function must be called exactly once in the constructor of the toplevel module.

The OSCI simulator does not support global asynchronous resets.

Definition

`void ag_global_async_reset_is(sc_in<bool> &`***port***`, bool` ***polarity***`);`

The function takes two arguments:

- a reference to an `sc_in<bool>` which is the input port for the reset signal
- a bool indicating the polarity of the reset.

When the input port changes to the given polarity value, the whole circuit is asynchronously reset, including resetting FIFOs and re-initializing all registers.

Example

```
class some_module : public sc_module
{
    sc_in<bool> clock_in;
    sc_in<bool> async_reset_in;

    void run();
public:
    SC_CTOR(some_module)
    {
        SC_THREAD(run);
        sensitive << clock_in.pos;
        ag_global_async_reset_is(async_reset_in, true);
    }
};
```

2.11.4 Initialization

Agility initializes instances of the following types to zero to match simulation behaviour:

- signals (`sc_signal`)
- output ports (`sc_out`, `sc_out_clk`)
- SystemC integer types (`sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint`)
- `sc_bv`

Often initializers do not affect the functionality of the design and are therefore not necessary. However, they may interfere with certain optimizations performed by Agility.

For example, if a registered variable is only ever written to once, Agility can replace it by a constant. If the register is initialized and then written to once, Agility cannot replace it with a constant.

Uninitialized registers can improve the results of retiming drastically, as initializers restrict registers from being moved through logic boundaries, making it harder to balance logic evenly across pipeline stages.

ag_undefined initialization

Signals, output ports, `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint` and `sc_bv` have an additional constructor to create uninitialized instances. This constructor takes one argument of type ***ag_undefined_type***, which is defined by the pre-processor macro `AG_UNDEFINED`.

Example

```
#include "systemc.h"
#include "agility.h"

SC_MODULE( TestInit )
{
  sc_in<bool> Clock;
  sc_out< sc_uint<32> > ResultA;
  sc_out< sc_uint<32> > ResultB;

  sc_signal< sc_uint<32> > MySigA;
  sc_signal< sc_uint<32> > MySigB;

  SC_CTOR( TestInit ) :
     ResultA( AG_UNDEFINED ), // don't initialize ResultA
     MySigA( AG_UNDEFINED )   // don't initialize MySigA
  {
    SC_THREAD( test );
    sensitive_pos << Clock;
  }

  void test()
  {
    sc_uint<32> MyVarA ( AG_UNDEFINED ); // don't initialize MyVarA
    sc_uint<32> MyVarB;

    wait();
      // ResultA = undefined
      // ResultB = 0
    wait();
    ResultA = MyVarB; // ResultA = 0
    ResultB = MyVarA; // ResultB = undefined
    wait();
    ResultA = MySigA.read(); // ResultA = undefined
    ResultB = MySigB.read(); // ResultB = 0
    wait();
  }
};
```

Global and thread initializers

Variables can be initialized either during elaboration (globally or inside the module constructor) or inside a thread. They can be initialized in both stages, but only if the initializers are identical. If this is not the case, an error (E2082) is produced. For example:

```
SC_MODULE( my_module )
{
  sc_in<bool> Clock;

  // initialized to '0' by default
  sc_out< sc_uint<32> > Result;

  void my_thread();

  SC_CTOR( my_module )
  {
    SC_THREAD ( my_thread );
    sensitive << clock.pos();
  }
}
void my_module::my_thread()
{
  // initialized to '2', error!
  Result = 2;
  wait();
  ...
}
```

In this example, an error will be produced as `Result` is initialized to `0` during elaboration and `2` inside the thread.

ag_override_global_init

To ensure that a variable is initialized with the value specified in a thread, use the constraint function `ag_override_global_init`. This allows you to specify that the initializers inside a thread override any initializations performed during elaboration.

```
template<typename T>
ag_override_global_init( const T &the_var );
ag_override_global_inits()
```

Use `ag_override_global_init` to apply thread initialization override to ***the_var.***

Use `ag_override_global_inits` to apply thread initialization for all variables in a design.

`ag_override_global_init(s)` may be called either inside the entry-point module (`ag_main` or `sc_main)` or the module constructor. For example:

```
SC_MODULE( my_module )
{
  sc_in<bool> Clock;

  // initialized to '0' by default
  sc_out< sc_uint<32> > Result;

  void my_thread();

  SC_CTOR( my_module )
  {
    SC_THREAD ( my_thread );
    sensitive << clock.pos();

    ag_override_global_init( Result );
  }
}
void my_module::my_thread()
{
  // initialized to '2', overriding '0'
  Result = 2;
  wait();
  ...
}
```

2.11.5 Assertions

Assertions are conditions that a designer expects to be true at certain points in a design. They are useful for documenting and validating code.

Agility supports *compile-time assertions*; that is, assertions that can be checked as the design is compiled. The functions provided check if conditions are known at compile-time, as well as whether the conditions are true.

- `void ag_assert_known(bool` ***Condition***`, char *`***Msg***`);`
  Checks if ***Condition*** is known at compile-time.
- `void ag_assert_known_and_true(bool` ***Condition***`, char *`***Msg***`);`
- `void ag_assert_unknown_or_true(bool` ***Condition***`, char *`***Msg***`);`
  These functions are used to test the value of ***Condition.*** `ag_assert_known_and_true` fails if ***Condition*** is unknown, whereas `ag_assert_unknown_or_true` passes.

Agility does not support run-time assertions.

ag_assert_known

`void ag_assert_known(bool` ***Condition,*** `const char *` ***Message);***
This checks to see whether ***Condition*** provided as argument evaluates to a boolean value at compile time.

Example

```
int y = 0;
for (int i = 0; i < 10; I++)
{
      y += sig.read();
      Result = y+1;
      wait();
}


ag_assert_known( y > 10, "y is not known at compile-time");
```

Agility is unable to infer the value of `y`, as its mechanism for inferring compile-time values (the ***loop unroller*** (see page 116)) skips over loops that contain a `wait` in the control path.

ag_assert_known_and_true

`void ag_assert_known_and_true(bool` ***Condition,*** `const char *` ***Message);***
Asserts that the value of ***Condition*** is known and is true at compile-time. If it is not known or is false then ***Message*** is displayed and the compilation stops.

Example
Consider the following way of summing the numbers `1`,`2`,`3`... up to `x`, where `x` is greater than or equal to zero:

```
int sum( int x )
{
     switch (x)
     {
        case 0: return 0
        default: return (1 + sum(x – 1));
     }
}
```

Calling `sum` with `10` returns `55`. The definition fails to terminate if the argument to `sum` is negative. You can use `ag_assert_known_and_true` to document this, if `x` is a compile-time constant.

```
int sum( int x )
{
     ag_assert_known_and_true( x >=0, "x is negative or unknown");
     switch (x)
     {
        case 0: return 0
        default: return (1 + sum(x – 1));
     }
}
```

ag_assert_unknown_or_true

`void ag_assert_unknown_or_true(bool` ***Condition***`, const char *` ***Message***`);`
Asserts that the value of ***Condition*** is either unknown or known and true. If it is known and false then ***Message*** is displayed and the compile stops.

Example

```
ag_assert_unknown_or_true(a != 0, "The value of a has been determined to be
zero.");
```

User-defined errors, warnings and information messages

Agility allows you to define your own errors, warnings and information messages. The message to be output by Agility is provided as a string argument:

- `void ag_error ( char *`***Msg*** `);`
- `void ag_warning( char *`***Msg*** `);`
- `void ag_information( char *`***Msg*** `);`

These will behave as standard Agility errors, warnings and information messages.

For example:

```
if  ( x > 2 )
{
    ... // some synthesizable code
}
else
{
     ag_error  ( " x should be greater than 2" );
     ...
}
```

Restrictions

`ag_error`, `ag_warning` and `ag_information` should not be used in:

- `if` statements whose condition is not compile-time determinable
- loop bodies containing a `wait()` statement in the control-path.

If an `ag_error` call is used in one of these situations it will be invoked whenever the code is compiled, irrespective of the condition, and even if the code is unreachable.

2.11.6 Signal with default

Agility provides the signal variant `ag_signal_default`, which takes an initial value. When you assign to this signal, it holds the assigned value for one clock cycle. The signal takes on its initialization value at all other times.

This behaviour is useful for control signals, as it ensures that their value does not remain from a previous operation. Using signals with a default means that you do not have to write to this control signal on clock cycles when the resource is not used.

For example, suppose you want to connect a write enable signal to a memory resource or bus. To perform a write you set this signal to 1; it should be 0 at all other times.

If an ordinary signal is used, you must explicitly reset this signal to zero after the resource access. However, if you use an `ag_signal_default`, this is done automatically when the signal is not written to.

Agility also provides a simulation model of `ag_signal_default` with identical behaviour.

Template parameters

`ag_signal_default` takes one template parameter: the type of the signal.

Constructor

`ag_signal_default( sc_module_name` ***name,*** `const` ***T*** `&` ***Default*** `)`

Construct initialized signal with instance name *name* (used for simulation) and default value *Default*.

Accessing an ag_signal_default

`ag_signal_default` has three member functions, `read`, `write` and `Clock`.

The `read` and `write` member functions are identical to the standard `sc_signal` member functions.

The `Clock` member function is used to assign a clock to synchronize the default.

`Clock(sc_in<bool>` ***Clock_name***`);`

E.g., `MySig.Clock(Clk);`

You must call the `Clock` function in the constructor of the module in which the signal is declared.

Port mapping

An `ag_signal_default` can be bound to standard `sc_in`, `sc_out` and `sc_inout` ports in exactly the same way as an `sc_signal`.

Example: ag_signal_default

```
#include "systemc.h"
#include "agility.h"

SC_MODULE( myModule )
{
  sc_in<bool> Clk;
  sc_out< sc_uint<32> > Result;

  /*
   * Declaration of signal with default
   */
  ag_signal_default< sc_uint<32> > mySig;

  SC_CTOR( myModule ) :
    mySig( "mySignal", 5 ) // construct signal with default set to 5
  {
    /*
     * Assign clock to the signal
     */
    mySig.Clock(Clk);

    SC_THREAD( test1 );
    sensitive_pos << Clk;

    SC_THREAD( test2 );
    sensitive_pos << Clk;
  }

  void test1()
  {
    wait();
    // mySig is not assigned to in this cycle, thus mySig = 5
    wait();
    mySig.write( 10 ); // mySig = 10
    wait();
    // mySig = 5
    wait();
    mySig.write( 20 ); // mySig = 20
    wait();
    // mySig = 5
  }

  void test2()
  {
    wait();
    while(1)
    {
```

```
      Result = mySig.read();
      wait()
    }
  }
```

2.12 Language features and restrictions summary

This section gives a brief overview of the SystemC features supported by Agility.

2.12.1 Supported SystemC language features

- Black box component instantiation
- Clocked thread processes (`SC_CTHREAD`)
- Custom ports
- Hierarchical channels using `sc_interface`
- Hierarchical modules
- Mapping multiple `sc_interface`s to one `sc_port`
- Method processes (`SC_METHOD`)
- Modules
- Multiple clock domains
- Multiple threads in a module
- Port mapping
- Reset sensitivity
- `sc_signal`
- Single-edge clock sensitivity
- Terminating threads
- Thread processes (`SC_THREAD`)
- `wait()` statement

Not supported for synthesis
The following features are not supported for synthesis

- Double-edge clock sensitivity
- Sensitivity to signals other than the clock in thread processes
- Virtual inheritance
- `wait(`***n***`)` and `wait_until()`

2.12.2 Supported SystemC/C++ types

Arrays, including arrays of complex types and multidimensional arrays

- `bool`
- `char`, `short`, `long`, `long long` (signed and unsigned)

- `float, double`
- `sc_bigint, sc_biguint`
- `sc_bit`
- `sc_bv`
- `sc_fifo`
- `sc_fixed` (see page 62)
- `sc_in, sc_out, sc_inout`
- `sc_inout_rv`
- `sc_int, sc_uint`
- `sc_logic`
- `sc_lv` (with restrictions)
- `sc_signal`
- Classes

  Only `const static` data members are supported

SystemC and C++ types and operators not supported for synthesis

- Arrays of pointers
- `float` and `double` can only be used for compile-time determinable values
- Increment and decrement operators are not supported for operands of type `bool`
- Integral types are represented using 2's complement
- Pointer arithmetic
- `sc_fifo` members `num_free()` and `num_available()` are not supported
- Unions
- `sc_ufixed`

3 Maths library

Agility provides a maths library for use at compile-time.

This allows you to use pre-defined functions to calculate cosine/sine/tangent lookup tables or other such complex calculations.

3.1 Agility maths library functions

The Agility compiler supports the C/C++ maths functions for compile-time evaluation. These functions are not themselves synthesizable. Their arguments must be compile-time determinable.

For example

```
float x = 0.756;
wait();
int result = sinh(x);
wait();
int result = cos(x);
```

This is synthesizable as the results of `sinh(x)` and `cos(x)` are cast to integers.

3.2 Header

The library may be included in your code in the normal C++ way:

```
#include <cmath>  // includes code in std
                  // namespace
using namespace std;
```

or in the C style:

```
#include <math.h>
```

The C++ style is recommended. The C style is provided for backward compatibility with legacy code.

3.3 Double precision ANSI C functions

`double acos(double` ***cs***`);`

Returns the arc cosine of ***cs*** (in radians).

If the value is too small `acos()` returns `0.0`, but whether `errno` is set to `ERANGE` is implementation dependent.

`double asin(double` ***sn***`);`

Returns (in radians) the arc sine of ***sn*** (in radians).

`double atan(double` ***tn***`);`

Returns (in radians) the arc tangent of ***tn*** (in radians).

`double atan2(double` ***sn,*** `double` ***cs*****);**

Returns the value of the arc tangent of ***sn*** divided by ***cs,*** using the signs of both arguments to calculate its quadrant. The angle returned is the anti-clockwise angle measured from the x-axis (in radians).

`double ceil(double` ***val*****);**

Returns the value of ***val***, rounded up to the next whole number.

`double cos(double` ***angle*****);**

Returns the cosine of ***angle*** in radians.

`double cosh(double` ***val*****);**

Returns the hyperbolic cosine of ***val*** in radians.

`double exp(double` ***val*****);**

Returns a double precision value containing ***e*** to the power of ***val.***

`double fabs(double` ***val*****);**

Returns the absolute (positive) value of ***val.***

`double floor(double` ***val*****);**

Returns the value of ***val***, rounded down to the next whole number.

`double fmod(double` ***a,*** `double` ***b*****);**

Returns the remainder of ***a*** divided by ***b.***

`double frexp(double` ***val,*** `int *`***exp*****);**

Returns the mantissa of ***val***, and stores the exponent in the integer pointed to by ***exp.***

`double ldexp(double` ***val,*** `int` ***exp*****);**

Returns the value of ***val*** multiplied by (2 to the power of ***exp).***

`double log(double` ***val*****);**

Returns the natural logarithm of ***val***.

`double log10(double` ***val*****);**

Returns the base 10 logarithm of ***val***.

`double modf(double` ***val*****,** `double *`***int_val***`);`

Returns the signed fractional component of ***val***, and places its whole number component into the double pointed to by ***int_val***.

`double pow(double` ***base*****,** `double` ***exp***`);`

Returns the value of ***base*** raised to the power of ***exp***.

`double sin(double` ***angle***`);`

Returns the sine of ***angle*** in radians.

`double sinh(double` ***val***`);`

Returns the hyperbolic sine of ***val*** in radians.

`double sqrt(double` ***squared***`);`

Returns the square root of ***squared.***

`double tan(double` ***angle***`);`

Returns the tangent of ***angle*** in radians.

`double tanh(double` ***val***`);`

Returns the hyperbolic tangent of ***val*** in radians.

3.4 Single precision ANSI C functions

`float acos(float` ***cs***`);`

Returns the arc cosine of ***cs*** (in radians).

If the value is too small `acos()` returns 0.0. No error values are set.

`float asin(float` ***sn***`);`

Returns (in radians) the arc sine of ***sn***.

`float atan(float` ***tn***`);`

Returns (in radians) the arc tangent of ***tn***.

`float atan2(float` ***sn*****,** `float` ***cs***`);`

Returns the value of the arc tangent of ***sn*** divided by ***cs*****,** using the signs of both arguments to calculate its quadrant. The angle returned is the anti-clockwise angle measured from the x-axis (in radians).

`float ceil(float` ***val***`);`

Returns the value of ***val***, rounded up to the next whole number.

`float cos(float` ***angle***`);`

Returns the cosine of ***angle*** in radians.

`float cosh(float` ***val***`);`

Returns the hyperbolic cosine of ***val*** in radians.

`float exp(float` ***val***`);`

Returns a float precision value containing ***e*** to the power of ***val.***

`float fabs(float` ***val***`);`

Returns the absolute (positive) value of ***val.***

`float floor(float` ***val***`);`

Returns the value of ***val***, rounded down to the next whole number.

`float fmod(float` ***a***`, float` ***b***`);`

Returns the remainder of ***a*** divided by ***b.***

`float frexp(float` ***val***`, int *`***exp**`);`

Returns the mantissa of ***val***, and stores the exponent in the integer pointed to by ***exp.***

`float ldexp(float` ***val***`, int` ***exp***`);`

Returns the value of ***val*** multiplied by (2 to the power of ***exp).***

`float log(float` ***val***`);`

Returns the natural logarithm of ***val***.

`float log10(float` ***val***`);`

Returns the base 10 logarithm of ***val***.

`float modf(float` ***val***`, float *`***int_val**`);`

Returns the signed fractional component of ***val***, and places its whole number component into the float pointed to by ***int_val***.

`float pow(float` ***base***`, float` ***exp***`);`

Returns the value of ***base*** raised to the power of ***exp***.

`float sin(float` ***angle***`);`

Returns the sine of ***angle*** in radians.

`float sinh(float` ***val***`);`

Returns the hyperbolic sine of ***val*** in radians.

`float sqrt(float` ***squared***`);`

Returns the square root of ***squared.***

`float tan(float` ***angle***`);`

Returns the tangent of ***angle*** in radians.

`float tanh(float` ***val***`);`

Returns the hyperbolic tangent of ***val*** in radians.

4 Agility synthesis guide

This section describes the Agility synthesis process and what is synthesized by Agility.

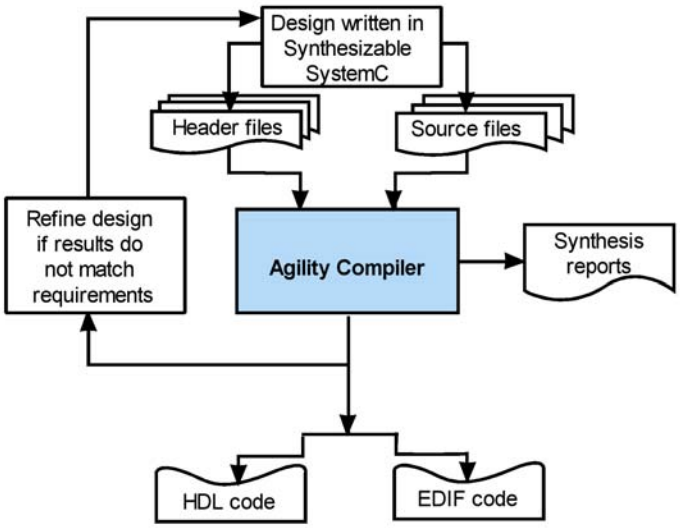
4.1 Agility design flow

Agility is a tool for synthesizing SystemC to EDIF or RTL HDL (Verilog, VHDL, SystemC).

It goes through the following stages:

- Preprocess
- Compile
- Link
- Compile-time evaluation
- ***High-level synthesis*** (see page 120)
- ***Optimization*** (see page 124)

Output files are then produced in SystemC RTL, EDIF, VHDL or Verilog.



4.1.1 Design flow stages

The expected design flow stages are given below. You may need to add or omit stages according to your working practices:

1. Produce a simulation model of your design in SystemC/C++.
2. Produce a testbench (also in SystemC/C++).
3. Simulate and test the model.
4. Refine the simulation design into synthesizable code.
   You will have to carry out this refinement manually.

Refer to the ***Agility SystemC Synthesizable Subset Reference Manual*** (see page 31) to determine which parts of SystemC are synthesizable by Agility.

It is good practice to split your design into several files: it is common to place interfaces in header (`.h`) files and the corresponding implementations in `.cpp` files. This will allow you to recompile only those parts of a design you have changed.

5. Compile your design into RTL SystemC using Agility.
6. Test the synthesizable model.
7. Compile your design into HDL (EDIF, Verilog or VHDL) using Agility.

   Different synthesis flags and optimizations can be turned on during compilation.
8. Examine the synthesis reports produced, to see whether your synthesized circuit meets your design's required timing and area constraints.
9. If these are not met, refine your design and repeat the synthesis process. You may wish to add synthesis constraints.
10. Once the requirements are met, run the Place and Route tools. (Place and Route tools are not described in this manual. you should refer to the vendor-specific documentation for details.)

Agility code requirements
The source code and header files include:

- header files appropriate for simulation or synthesis (as specified in a preprocessor directive)
- an `ag_main` function, which is used for synthesis, and consists of an instantiation of a top-level module and any user constraints
- an `sc_main` function, which can be called for simulation, and consists of an instantiation of a top-level module and any user constraints
- a top-level module, which instantiates all modules to be synthesized
- the modules, threads and methods that describe the logic of your design
- The signals, FIFOs and other forms of communication (channels) which describe how the processes communicate

4.2 Constraints

Agility supports constraints that allow you to:

- direct the synthesis process
- affect the synthesis output.

The constraints do not alter the functionality of your design, but allow you to influence the size and speed of the resultant hardware without having to make changes to your code.

Constraints can be added:

- in `ag_main`,

  or
- within the constructors of your SystemC modules.

If you keep all your constraints within your `ag_main`, the file in which it is defined can be used as an independent constraints file, allowing you to perform design exploration without modifying any of your code.

Constraints available are:

- Synthesis constraints
  - Clock constraints
  - ***FIFO constraints*** (see page 93)
  - ***RAM constraints*** (see page 100)
  - ***Resource constraints*** (see page 112)
- ***Output constraints*** (see page 114)

4.2.1 Synthesis constraints

Synthesis constraints can be added either in `ag_main` or within the constructors of your SystemC modules.

Clock constraints

You can specify the period of a clock coming from a port or signal by using the clock constraint. This produces timing constraints with the EDIF output for the place and route tools.

```
ag_constrain_clock(
    sc_in<bool> & the_clock,
    enum sc_time_unit period_units,
    double period );

ag_constrain_clock(
    sc_signal<bool> & the_clock,
    enum sc_time_unit period_units,
    double period );
```

The clock is constrained by specifying a period and the units in which the period is expressed. `sc_time_unit` can take the following values:

- `SC_FS`: Femtoseconds (10^{-15}s)
- `SC_PS`: Picoseconds (10^{-12}s)
- `SC_NS`: Nanoseconds (10^{-9}s)
- `SC_US`: Microseconds (10^{-6}s)
- `SC_MS`: Milliseconds (10^{-3}s)
- `SC_SEC`: Seconds

When you specify a clock constraint together with the ***retiming option*** (see page 128), Agility moves flip-flops around in the circuit to try to meet the clock period specified.

Example: Clock constraints

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock;
    ...

    void Thread1();

    SC_CTOR( Impl )
    {
        SC_THREAD( Thread1, Clock.pos() );
    }
};

void ag_main()
{
    Impl TopLevel;

    // Port 'Clock' in TopLevel is specified to have a period
    // of 10 nanoseconds (rate of 100Mhz)
    ag_constrain_clock(
        TopLevel.Clock,
        SC_NS,
        10 );
}
```

FIFO constraints

When you use FIFOs to communicate between processes sensitive to different clocks, the timing of the synchronization hardware can be specified using the `ag_constrain_fifos_in_domain` function. It applies to both ports and signals:

```
ag_constrain_fifos_in_domain(
    sc_in<bool> & the_clock,
    const fifo_timing & timing,
    const unsigned metastability_protection_level);

ag_constrain_fifos_in_domain(
    sc_signal<bool> & the_clock,
    const fifo_timing & timing,
    const unsigned metastability_protection_level);
```

This function is applied to the clock and affects the timing of ***all*** channels to and from that clock domain.

Note that FIFOs will only be used if you have specified them using `sc_fifo`.

fifo_timing

`fifo_timing` is a structure passed to `ag_constrain_fifos_in_domain`. It is used to specify:

- `minimum_period`

OR

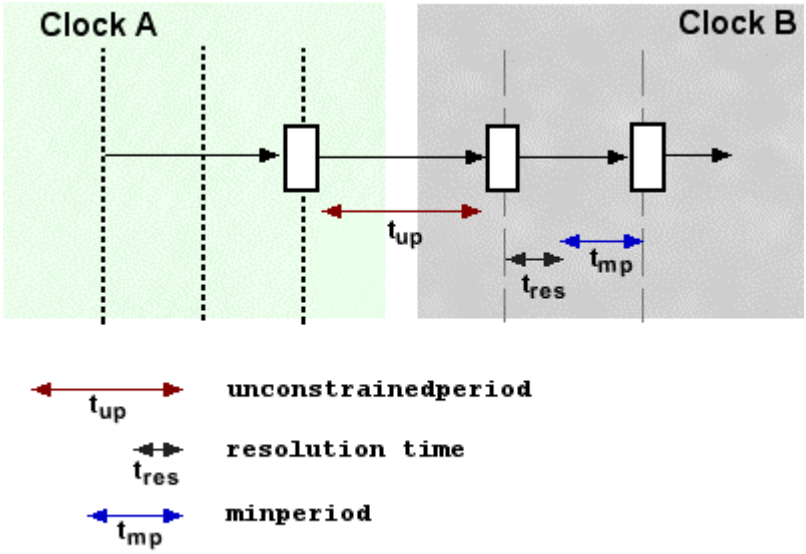
`resolution_time`

- `unconstrained_period`.

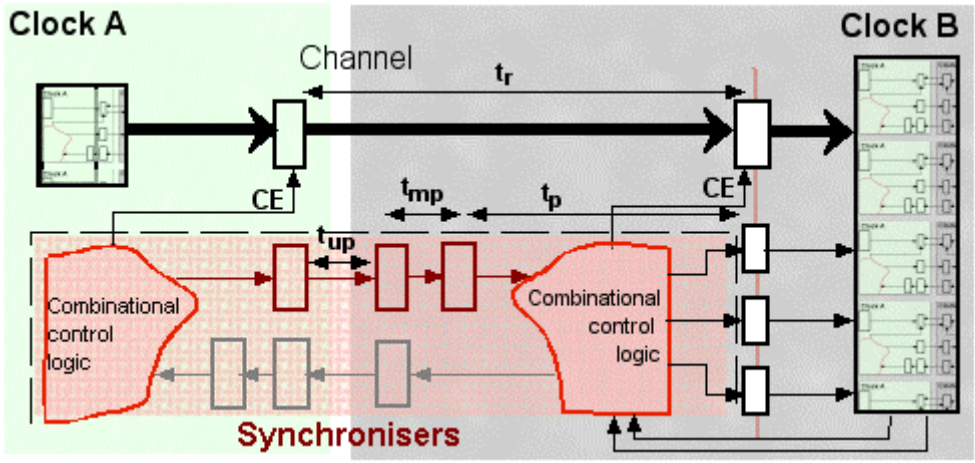
It only has an effect on FIFO timing when used as a parameter to `ag_constrain_fifos_in_domain`.

minimum_period constraint parameter

`minimum_period` specifies the maximum delay in nanoseconds between flip-flops in a synchronizer, including output delay, setup time and skew at either end). Its value must be less than the clock period.



The higher the value for `minimum_period`, the less time will be available within a clock tick for control signals to stabilize (`resolution_time`). You may set the value of `minimum_period` or `resolution_time`, but not both. If `metastability_protection_level` has been set to zero, you should use `minimum_period`.

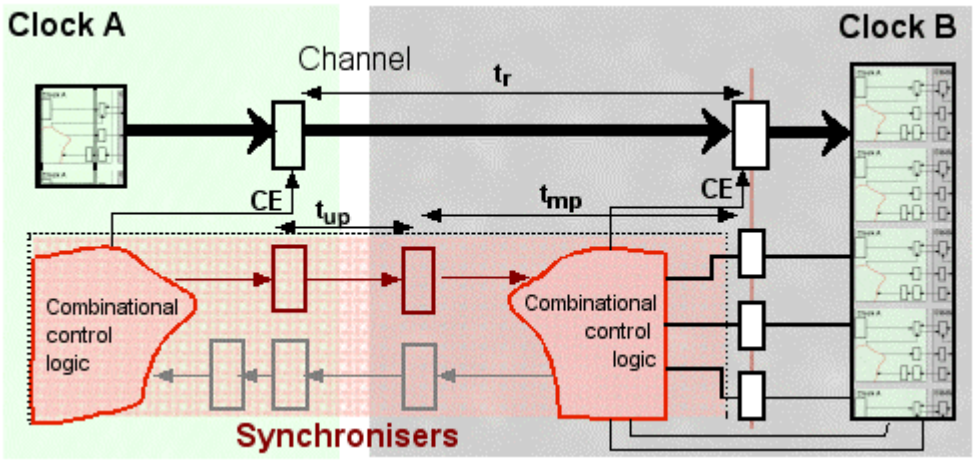


PERIODS WITH METASTABILITY_PROTECTION_LEVEL AT ITS DEFAULT OF 1

t_{mp} `minimum_period`

t_{up} `unconstrained_period`

t_p clock period

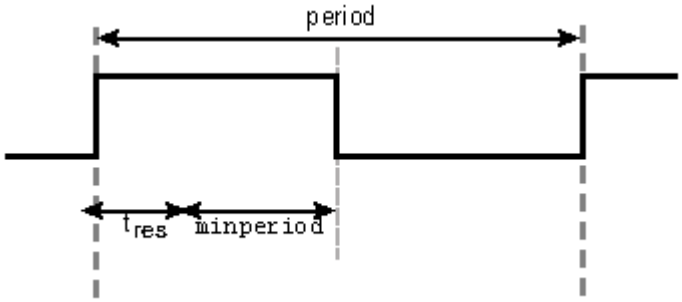


PERIODS IF METASTABILITY_PROTECTION_LEVEL SET TO 0

In this case, it is possible that the control signal may be metastable within the first flip-flop, and if `minimum_period` is inadequate, the metastability may be propagated into the rest of the circuit.

resolution_time constraint parameter

`resolution_time` specifies the maximum time in nanoseconds for metastability to resolve in the channel synchronization hardware. It is needed when you are using channels to communicate between multiple clock domains. The higher the value for `resolution_time` the less time will be available within a clock tick for combinational logic in the synchronizer. This only matters if you have set `metastability_protection_level` to zero.

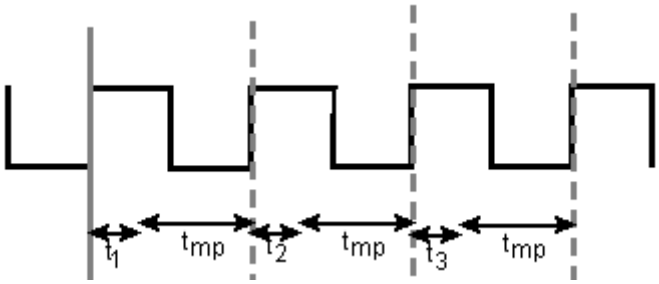


Its value must be less than (clock period x `metastability_protection_level)` where `metastability_protection_level` is greater than zero. If `metastability_protection_level` has been set to zero, you should use `minimum_period` rather than `resolution_time`.

Either `resolution_time` or `minimum_period` may be set, but not both.

Achieving a given value of resolution time

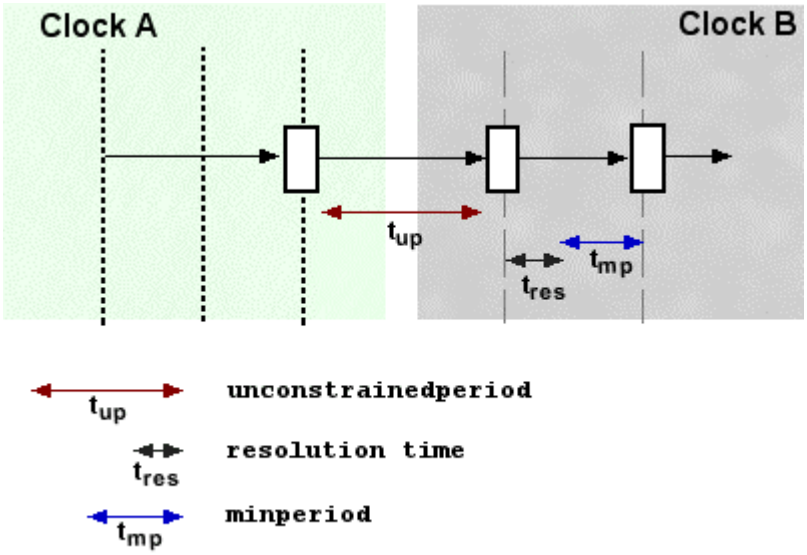
If you need a higher value of `resolution_time`, you can increase the value of the `metastability_protection_level` specification. The resultant value for `minimum_period` will be clock period - (`resolution_time`/`metastability_protection_level`).



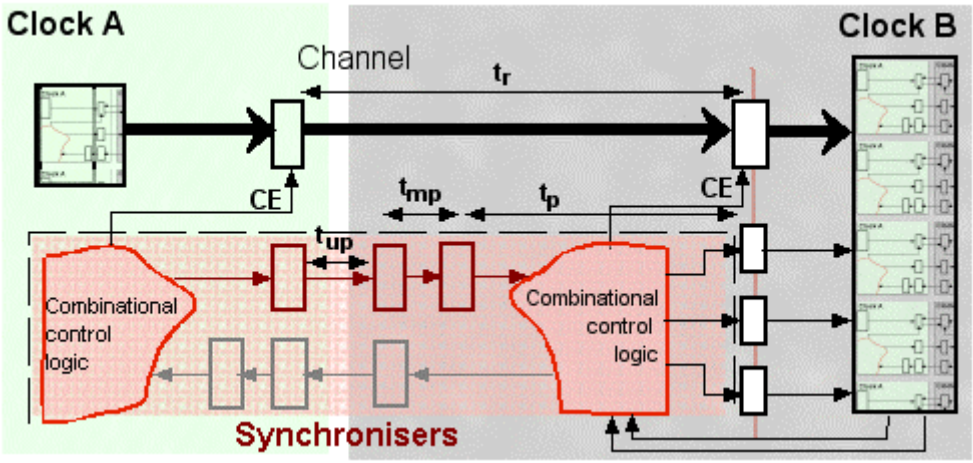
Resolution time summed over three clock ticks when `metastability_protection_level`= 3

unconstrained_period constraint parameter

`unconstrained_period` gives the maximum period in nanoseconds on channel control paths between clock domains. If this specification is not used, the place and route tools may generate a warning for affected paths crossing the clock domain.



If the specification is used, it applies to unconstrained paths into the clock domain. The diagram below shows where it is used.



TIMES WITH METASTABILITY_PROTECTION_LEVEL AT ITS DEFAULT OF 1

t_r: time to transfer between domains (`metastability_protection_level +1`) x t_p

t_{up}: `unconstrained_period`

t_{mp}: `minimum_period`

t_p: clock period

Specifying minimum_period and unconstrained_period
To specify `minimum_period`, use the following function to create an instance of `fifo_timing`:

```
fifo_timing::minimum_period(
    enum sc_time_unit period_units,
    double min_period,
    double unconstrained_period=-1.0 )
```

`sc_time_unit` can take the following values:

- `SC_FS`: Femtoseconds (10^{-15}s)
- `SC_PS`: Picoseconds (10^{-12}s)
- `SC_NS`: Nanoseconds (10^{-9}s)
- `SC_US`: Microseconds (10^{-6}s)
- `SC_MS`: Milliseconds (10^{-3}s)
- `SC_SEC`: Seconds

unconstrained_period is optional. If it is not supplied, the default value of -1.0 will be used.

Example

```
// Set minimum_period to 2 nanoseconds and unconstrained_period
// to 10 nanoseconds
ag_constrain_fifos_in_domain(
    MyClock,
    fifo_timing::minimum_period( SC_NS, 2, 10 ) );
```

Specifying resolution_time and unconstrained_period
You can specify `resolution_time` instead of `minimum_period` in a similar way:

```
fifo_timing::resolution_time(
    enum sc_time_unit period_units,
    double res_time,
    double unconstrained_period=-1.0 )
```

If ***unconstrained_period*** is not supplied, the default value of -1.0 will be used.

Example

```
// Set resolution_time to 7.5 ns, don't specify unconstrained_period
ag_constrain_fifos_in_domain(
    MyClock,
    fifo_timing::resolution_time( SC_NS, 7.5 ),
    2 );
```

metastability_protection_level constraint parameter
The last argument to `ag_constrain_fifos_in_domain` is `metastability_protection_level`, which controls the number of flip-flops used in synchronization hardware for FIFOs across clock domains. The higher the value for

`metastability_protection_level`, the higher the stability and latency of the FIFOs. The default is 1, which should be adequate in most cases. If latency is an issue, it is possible to set `metastability_protection_level` to 0, but the circuit is more likely to be metastable.

Example: FIFO constraints

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock1;
    sc_in<bool> Clock2;
    ...

    // Declare a FIFO to communicate between processes
    sc_fifo<sc_uint<32> > MyFIFO;

    void Thread1();
    void Thread2();

    SC_CTOR( Impl )
    {
        SC_THREAD( Thread1, Clock1.pos() );
        SC_THREAD( Thread2, Clock2.pos() );
    }
};

void ag_main()
{
    Impl TopLevel;

    // Port 'Clock1' in TopLevel is specified to have a period
    // of 10 nanoseconds (rate of 100Mhz)
    ag_constrain_clock( TopLevel.Clock1, SC_NS, 10 );

    // Port 'Clock2' in TopLevel is specified to have a period
    // of 20 nanoseconds (rate of 100Mhz)
    ag_constrain_clock( TopLevel.Clock1, SC_NS, 20 );

    // minimum_period is specified to be 2 nanoseconds
    // unconstrained_period is specified to be 10 nanoseconds
    // metastability_protection_level is specified to be 2
    ag_constrain_fifos_in_domain(
        MyClock1,
        fifo_timing::minimum_period( SC_NS, 2, 10 ),
        2 );

}
```

RAM constraints

In Agility, arrays can be implemented as RAMs. This allows you to make use of dedicated RAM resources on your target architecture. RAMs are introduced using the `ag_constrain_ram` constraint:

```
template<typename T>
ag_constrain_ram(
    const T & the_array,
    enum ag_ram_access_type read_access );
```

`ag_constrain_ram` must be called from either `ag_main` or a module constructor and cannot be called inside a process. This is true of all the constraints associated with RAMs.

RAM constraints constrain a member variable of a particular instance of a class. The constraints only apply to that instance of the RAM in that instance of the class.

`ag_constrain_ram` takes two arguments:

the_array

the array to be implemented as a RAM

read_access

whether the RAM has synchronous or asynchronous read capability. ***read_access*** can be either:

- `ag_ram_synchronous`
- `ag_ram_asynchronous`

When targeting an FPGA, Agility automatically uses the appropriate RAM resource with the specified read capability.

E.g. block RAM for synchronous and distributed RAM for asynchronous.

`ag_constrain_ram` can be applied to arrays of simple data types as well as complex types such as `struct`s. It is ignored in simulation.

Agility

RAMs versus arrays

When an array is implemented as a RAM, the behaviour of the design may change due to differences in timing semantics between the two. For example, take the following SystemC:

```
sc_uint<32> MyArray[32];
      ....
sc_uint<32> MyVar;
wait();
MyArray[2] = 3;
MyVar = MyArray[2];
wait();
```

In this case, the array is written to and subsequently read from the same location in the same clock cycle.

If `MyArray` is a simple array, the value that is written will be available in the same cycle and the value of `MyVar` will be 3 at the end of the cycle.

If `MyArray` is implemented as a dual-port RAM, the value that is read into `MyVar` will be undefined as `MyArray[2]` is accessed from multiple ports simultaneously.

RAM restrictions

The following restrictions apply to RAMs in Agility:

- dynamically-indexed arrays of RAMs are not supported
- `ag_constrain_ram` can only be applied to entire arrays, not partial arrays.

  For example, in an array of arrays, the entire array would have to be implemented as a RAM; the individual array elements (which are arrays) could not be specified as RAMs.

Example: RAM

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock;
    sc_out<sc_uint<8> > Result;
    sc_in<sc_uint<8> > Input;

    // Declare RAM
    sc_uint<8> MyRam[8];

    void Thread();

    SC_CTOR( Impl )
    {
        SC_CTHREAD( Thread, Clock.pos() );

        // Declare MyRam as a memory with asynchronous read access
        ag_constrain_ram( MyRam, ag_ram_asynchronous );

        // Initialise the RAM
        for( int i=0; i<8; i++ )
        {
            MyRam[i] = 0;
        }
    }

    void Thread()
    {
        sc_uint<3> Index;

      wait();
      while ( 1 )
      {
        MyRam[Index] = Input.read();
        wait();
        Result = MyRam[7-Index];
        Index++;
        wait();
      }
    }
};
```

Agility will automatically check whether a RAM is accessed more than once in the same clock cycle and produce an error if this is the case.

Memory ports

By default, `ag_constrain_ram` declares a single-port RAM with one read-write port. To declare a ROM or a multi-port RAM, memory ports can be added using the `ag_add_ram_port` constraint:

```
ag_add_ram_port(
    const T & the_array,
    enum ag_ram_port_type type,
    const char *name )(pointer_to_thread)[(pointer_to_thread)];
```

type specifies the access mode of the port, and takes one of the following values:

- `ag_port_readonly`
- `ag_port_writeonly`
- `ag_port_readwrite`

name must be a unique name.

pointer_to_thread specifies a thread that accesses the port. Multiple threads can be specified in the form `(pt_Thread1)(pt_Thread2)(pt_ThreadN)`.

You can only call `ag_add_ram_port` on an array that has been declared as a RAM using `ag_constrain_ram`.

ROMs

To declare a single-port ROM, declare a RAM and add one read-only port.

Example: using a ROM to implement a sine table

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock;
    sc_out<sc_int<16> > Result;

    // Declare ROM
    sc_int<16> SineTable [128];

    void Thread();

    SC_CTOR( Impl )
    {
        SC_CTHREAD( Thread, Clock.pos() );

        // Declare SineTable as a memory with asynchronous
        // read access
        ag_constrain_ram( SineTable, ag_ram_asynchronous );

        // Add single read port
        ag_add_ram_port( SineTable, ag_port_readonly, "ReadPort"
)(&Impl::THread);

        // initialize the sine table
        for ( int i = 0; i < 128; i++ )
  {
    double index = (double) i;
    double angle = 2.0 * PI * (index / 128.0);
    double sineangle = sin( angle );
    SineTable[i] = (32767.0 * sineangle);
  }
    }

    void Thread()
    {
      wait();
      for ( int i=0; i<128; i++ )
      {
        Result = SineTable[i];
        wait();
      }
    }
};
```

Multi-port RAMs

To declare a multi-port RAM, declare a RAM and add multiple ports. The RAM is still accessed as a normal array and Agility will automatically map each RAM access to an available port.

Example: Declaring a synchronous dual-port RAM

The code below declares a synchronous RAM with one read and one write port:

```
#include "systemc.h"
#include "agility.h"
SC_MODULE( Impl )
{
    sc_in<bool> Clock;
    sc_out<sc_uint<8> > Result;
    sc_in<sc_uint<8> > Input;
    // Declare RAM
    sc_uint<8> MyRam[8];
    void Thread1()
    {
        sc_uint<3> Index;
        wait();
        while ( 1 )
        {
            MyRam[Index] = Input.read(); // mapped to
"WritePort"
            Index++;
            wait();
        }
    }
    void Thread2()
    {
        sc_uint<3> Index;
        wait();
        while ( 1 )
        {
            Result = MyRam[7-Index]; // mapped to "ReadPort"
        Index++;
        wait();
        }
    }
    SC_CTOR( Impl )
    {
        SC_CTHREAD( Thread1, Clock.pos() );
        SC_CTHREAD( Thread2, Clock.pos() );
        // Declare MyRam as a memory with synchronous read access
        ag_constrain_ram( MyRam, ag_ram_synchronous );
        // Add read-only port accessed by Thread2
        ag_add_ram_port( MyRam, ag_port_readonly, "ReadPort" )(
&Impl::Thread2 );
        // Add write-only port accessed by Thread1
        ag_add_ram_port( MyRam, ag_port_writeonly, "WritePort" )(
&Impl::Thread1 );
        // Initialise the RAM
        for( int i=0; i<8; i++ )
        {
            MyRam[i] = i;
        }
    }
};
```

Multiple access error

If Agility is not able to find an available port for a RAM access, an error is reported. This can happen if a memory is accessed multiple times in the same clock cycle and does not have enough ports.

For example:

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock;
    sc_out<sc_uint<8> > Result;
    sc_in<sc_uint<8> > Input;

    // Declare RAM
    sc_uint<8> MyRam[8];

    void Thread();

    SC_CTOR( Impl )
    {
        SC_THREAD( Thread, Clock.pos() );

        // Declare MyRam as a single-port RAM with
        // synchronous read access
        ag_constrain_ram( MyRam, ag_ram_synchronous );

        // Initialise the RAM
        for( int i=0; i<8; i++ )
        {
            MyRam[i] = 0;
        }
    }

    void Thread()
    {
        sc_uint<3> Index;

        wait();
        while ( 1 )
        {
            // mapped to default read-write port
            MyRam[Index] = Input.read();

            // Error: no available port for this access
            Result = MyRam[7-Index];

            Index++;
            wait();
        }
    }
};
```

Black box RAMs

If the implementation of the RAM is not defined in Agility, it can be instantiated as a black box using the `ag_ram_as_blackbox` constraint:

```
ag_ram_as_blackbox(
    const T & the_array,
    const char * name );
```

This function is passed an array and the name of the component to be instantiated.

You can only call `ag_ram_as_blackbox` on an array that has been constrained to be a RAM.

Example: Dual-port RAM declared as a black box

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock;
    sc_out<sc_uint<8> > Result;
    sc_in<sc_uint<8> > Input;

    // Declare RAM
    sc_uint<8> MyRam[8];

    void Thread();

    SC_CTOR( Impl )
    {
        SC_CTHREAD( Thread, Clock.pos() );

        // Declare MyRam as a memory with synchronous read access
        ag_constrain_ram( MyRam, ag_ram_synchronous );

        // Add read-only port
        ag_add_ram_port( MyRam, ag_port_readonly, “ReadPort”
)(&Impl::Thread);
        // Add write-only port
        ag_add_ram_port( MyRam, ag_port_writeonly, “WritePort”
)(&Impl::Thread);

        // Declare the RAM as a black box
        ag_ram_as_blackbox( MyRam, “MyComponent” );

        // Initialise the RAM
        for( int i=0; i<8; i++ )
        {
          MyRam[i] = 0;
        }
    }
```

```
    void Thread()
    {
        sc_uint<3> Index;

        wait();
        while ( 1 )
        {
            MyRam[Index] = Input.read(); // mapped to "WritePort"
            Result = MyRam[7-Index]; // mapped to "ReadPort"
            Index++;
            wait();
        }
    }
};
```

This will produce an instance of the black box component `MyComponent` with appropriate memory control/data ports.

Pipelining synchronous memories

In SystemC semantics, an array access always takes no more than one clock cycle. However, synchronous RAM reads take two cycles: one to set up the address and one to read in the data. To accomplish this, Agility tries to pipeline memory accesses. If Agility is not able to do this, synchronous memories are clocked using an inverted version of the domain clock. This results in designs with tight time constraints, in the worst case halving their maximum clock frequency. It is therefore desirable to pipeline synchronous RAMs wherever possible.

For Agility to perform pipelining, you need to use the coding style described below.

Coding style for pipelining at the output

Pipelining synchronous RAM at the output means that Agility will attempt to absorb registers that are fed by the RAM. For memory accesses to be pipelined, the following conditions must be satisfied:

Each RAM port must be read into a single uninitialized register and nowhere else.

Nothing else must write to this register.

Only FPGA architectures with RAMs supporting read-before-write mode can be pipelined at the output.

Example: Pipelining at the output of a single-port RAM

```
#include "systemc.h"
#include "agility.h"

SC_MODULE( test_module )
{
    // clock port
    sc_in<bool> Clock;

    sc_out<sc_uint<32> > O;
    sc_in<sc_uint<32> > I;


    sc_uint<32> TheRam[32];
public:
    void myThread();

    SC_CTOR( test_module )
    {
        // declare single-port memory
        ag_constrain_ram( TheRam, ag_ram_synchronous );
        SC_THREAD( myThread );
        sensitive_pos << Clock;
    }
};


void test_module::myThread()
{
    sc_uint<5> index;

    // make sure RAM output data register is uninitialized
    sc_uint<32> x( AG_UNDEFINED );

    index = 0;

    wait();

    while(1)
    {
        TheRam[index] = I.read();
        wait();
        index++;
        wait();
        O = x;
        x = TheRam[index];
        wait();
    }
}
```

Example: Pipelining at the output of a dual-port RAM

```
#include "systemc.h"
#include "agility.h"

SC_MODULE( test_module )
{
    // clock port
    sc_in<bool> Clock;

    sc_out<sc_uint<32> > O1;
    sc_out<sc_uint<32> > O2;
    sc_in<sc_uint<32> > I1;
    sc_in<sc_uint<32> > I2;

    sc_uint<32> TheRam[32];
public:
    void myThread();

    SC_CTOR( test_module )
    {
        // declare dual-port memory
        ag_constrain_ram( TheRam, ag_ram_synchronous );
        ag_add_ram_port( TheRam, ag_port_readwrite, "rax1"
)(&test_module::myThread);
        ag_add_ram_port( TheRam, ag_port_readwrite, "rax2"
)(&test_module::myThread);

        SC_THREAD( myThread );
        sensitive_pos << Clock;
    }
};
```

```
void test_module::myThread()
{
    sc_uint<5> index1, index2;

    // make sure RAM output data registers are uninitialized
    sc_uint<32> x( AG_UNDEFINED );
    sc_uint<32> y( AG_UNDEFINED );

    index1 = 0;
    index2 = 1;

    wait();

    while(1)
    {
        TheRam[index1] = I1.read();
        TheRam[index2] = I2.read();
        wait();
        index1++;
        index2++;
        wait();
        O1 = x;
        x = TheRam[index1];
        O2 = y;
        y = TheRam[index2];
        wait();
    }
}
```

Resource constraints

You can limit the multipliers, adders, subtractors, and relational operators (<, <=, >, >=) available for a particular process by using the resource constraint directive. Resource constraints constrain a member function of a particular class (rather than an instance of a class). Thus the constraints apply to all threads that may be created from that member function and the syntax is ***Class*::*Member*** function.

```
ag_constrain_resource(
    void (T::* pointer_to_thread)(),
    enum ag_resource_type resource_type,
    int num_resources );
```

- Parameter ***pointer_to_thread*** is a pointer to the thread to which the resource constraint will apply.
- Parameter ***resource_type*** is an `enum` representing the resource type that is to be limited. Supported values are:

  `ag_resource_multiplier`

  `ag_resource_addsubcmp`

Using the enum `ag_resource_addsubcmp` limits the resources used for adders, subtractors and the relational operators (<, <=, >, >=).

Adders, subtractors and the relational operators are all implemented on "add/sub" ALU blocks. An add/sub can perform any of the following operations:

```
+, -, <, <=, >, >=
```

- Parameter ***num_resources*** is an integer representing the number of instances of the particular resource to be instantiated in the chosen thread. Setting this to zero will tell Agility to minimize the number of instances of the specified resource.

```
ag_constrain_resource(
        MyModule::Thread1,
        ag_resource_alu,
        3 );
```

limits the number of add/sub blocks used in `Thread1` of `MyModule` to 3. (The total number of the operations: +, –, <, <=, >, >=.)

You cannot apply a resource constraint to a module or a design, only to a process.

You can only have one constraint of each type on each process. However, you may put constraints on as many processes as you like.

```
//adder constraints on Thread1
ag_constrain_resource(
        MyModule::Thread1,
        ag_resource_alu,
        3);
//adder and multiplier constraints on Thread1
ag_constrain_resource(
        MyModule::Thread2,
        ag_resource_alu,
        3);
ag_constrain_resource(
        MyModule::Thread2,
        ag_resource_multiplier
        4);
```

Example: Resource constraints

This example shows resource constraints placed on a single thread via the `ag_main` function.

```
SC_MODULE( Impl )
{
    sc_in<bool> Clock;

    ...

    void Thread1();

    SC_CTOR( Impl )
    {
        SC_THREAD( Thread1, Clock.pos() );
    }
};

void ag_main()
{
    Impl TopLevel;

    // Thread1 from top-level must be synthesized using
    // only two multipliers
    ag_constrain_resource(
        Impl::Thread1,
        ag_resource_multiplier,
        2 );
}
```

4.2.2 Output constraints

Output constraints can be added to black box modules as well as to SystemC modules.

To change the type of a port in VHDL, use `ag_constrain_port`.

`ag_constrain_port ( ` ***port*** `, ag_constraint_vhdl_type, ` ***vhdl_type_string*** ` );`

where ***port*** is the port to constrain and ***vhdl_type_string*** is one of `signed`, `unsigned` or `std_logic_vector`.

Example

```
class MyModule : public sc_module
{
public:
  sc_in<bool> Clock;
  sc_in< sc_uint<2> > MyInput;
  sc_out< sc_uint<2> > MyOutput;

  MyModule()
  {
    ag_constrain_port( MyInput, ag_constraint_vhdl_type,
"std_logic_vector");
    ...
  }
}
```

will output port `MyInput` as `std_logic_vector`, rather than as `unsigned`. You can use the same construct to specify the type of ports on black boxes (i.e. instances of `ag_foreign_module` (see page 72)).

4.3 Agility synthesis

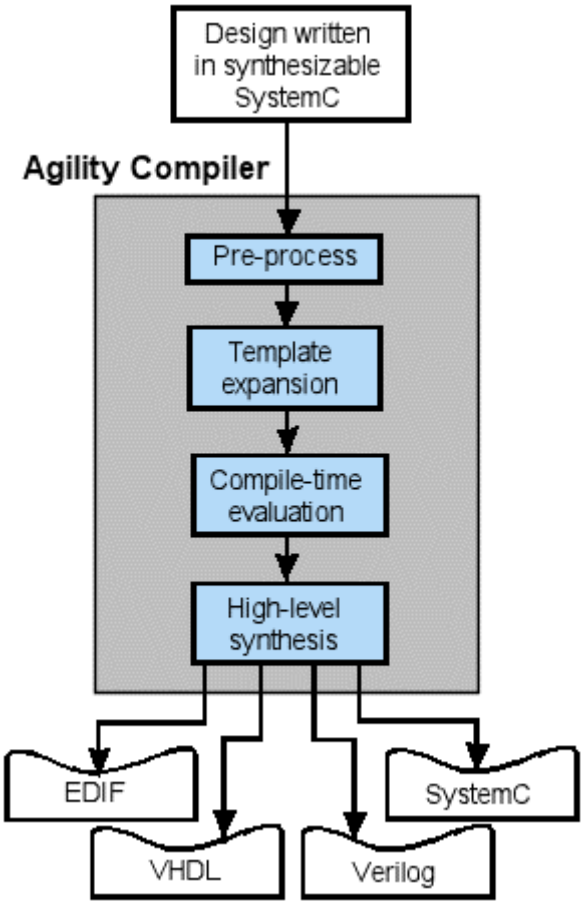
The Agility compiler goes through the following stages

Preprocess Compile Link	These three stages are equivalent to C++ compilation.

Compile-time evaluation

High-level synthesis (see page 120)

Optimization (see page 124)



4.3.1 Compile-time evaluation

Compile-time evaluation is performed after the link stage. At this stage, Agility unrolls loops, resolves pointers, inlines functions and works out all values of variables known at compile-time.

4.3.2 Loop unrolling

The loop unroller interprets loops that must be unrolled and skips over those that do not.

Any loops that do not contain `wait()` statements will be unrolled to produce combinational logic.

Example: Loop unrolling
This demonstrates loop unrolling with a combinational implementation of an FIR.

```
void process()
{
    char array[9];
    char coeffs[9] = {-1,8,-1,-2,4,-1,-2,0,3};

    // Wait for the first clock edge
    wait();
    // Execute this loop indefinitely
    while(1)
    {
        // Shift through the array of taps
        for (int i = 7; i >= 0; i--)
        {
            array[i+1] = array[i];
        }

        // Read a new input
        array[0] = input;

        // Combinational FIR
        result = array[0] * coeffs[0];
        for (int j = 1; j < 8; j++)
        {
            result += array[j] * coeffs[j];
        }
        // Wait for another clock edge
        wait();
    }
}
```

Loop unrolling limitations

The loop unroller cannot unroll loops containing:

- `wait` statements
- loop conditions including variables with unknown values.

Loops containing `wait()` statements are skipped and the values of variables in their bodies are tagged as ***unknown***.

If the condition statement of a loop that must be unrolled contains a variable with an ***unknown*** value, it causes an error.

You can sometimes avoid this problem by inserting assignments for such unknowns.

Example

```
int k = 0;
for(int i=0; i<10; i++)
{
   k = k + 5; //k increments to 45
   wait();
}

for (int j=0;j<k; j++)
{
   ... code without waits
}
```

The first loop is skipped because it contains a `wait` statement, leaving `k` set to ***unknown***. This means the second loop cannot be unrolled.

You can solve the problem by inserting an explicit assignment between the loops, as you know the value of `k` after the first loop has terminated.

```
for(int i=0; i<10; i++)
{
   k = k + 5;
   wait();
}

k = 45;  // explicit assignment to set k to the correct value
for (int j=0;j<k; j++)
{
   ... code without waits
}
```

4.3.3 Compile-time recursive definitions

Agility allows you to synthesize recursive definitions whose arguments can be determined at compile time.

You need to ensure that your recursive definitions terminate. For example, if you missed out a termination check in the example below `Fibonacci(5)` would not terminate.

Agility displays the number of times a function is called. This can help you track down a non-terminating function definition in your design.

Example: Fibonacci function
The function calculates the ***n***th Fibonacci number for any given ***n:***

```
int Fibonacci( int n )
{
     if (n == 0)       // check for termination
       return 0;
     if (n == 1)       // check for termination
       return 1;
     if (n > 1) return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

A call to `Fibonacci(6)` would be fully evaluated by Agility to 8.

4.3.4 Function inlining

Agility inlines all function definitions. For example, a call to this function:

```
int multiply_and_accumulate(int x, int y, int z)
{
   return (x * y + z);
}
```

would be replaced by the body of the function (suitably instantiated). If there are two calls to the function within the same clock cycle, two multipliers and two adders would be introduced.

The effect this has on the overall circuit size depends on the sharing that Agility implements.

4.3.5 Pointer resolution

Agility can synthesize pointers that are resolved at compile-time i.e. pointers that are not dynamically changed during the execution of the program.

Example

```
SC_MODULE(MyModule)
{
    DaughterModule * DaughterA;
    ...
    SC_CTOR(MyModule)
    {
        DaughterA = new DaughterModule();
        ...
    }
}
```

4.3.6 High-level synthesis

High-level synthesis is performed after the files have been compiled and linked. Agility analyzes the code to see what optimizations can be made.

Register inference

Registers are placed on every signal (and port) written to in a synchronous context.

Registers are inferred from the use of variables. For example:

```
int i = 1, j = 2, k;
for(;;)
{
    k = i + j;  // Clock cycle 1
    j++;
    i = k;
    wait();
    i++;        // Clock cycle 2
    wait();
}
```

- The value of `k` is only ever set and read on the same clock cycle and does not persist to another clock cycle: `k` is a wire.
- `i` is set and read in every clock cycle: `i` is a register.
- `j` is read in the first clock cycle and this value is read in the next execution of the loop: `j` is a register.

Loop conversion

Agility1.2 converts for and while loops to do-while loops to remove combinational logic.

For example the loop

```
for (i=0; i < 10; i++)
{
    for (j=0; j < 10; j++)
    {
        wait();
    }
}
```

will be converted to

```
do
{
    do
    {
        j++;
        wait();
    }while (j < 10);
    i++;
}while (i < 10);
```

except when the for/while loop condition contains side-effects (for example, an assignment or function call).

5 Loop optimisation

When Agility builds a run-time loop, variables read in the first cycle of the body could have been set either in the code immediately before the loop or after the previous cycle of a loop.

For example:

```
wait();
for ( int a = 0; a <= 4; a++ )
{
    wait();
}
```

expands to:

```
wait();
int a = 4;
while ( 1 )
{
    // The value of 'a' here can either be constant 4,
    // set before the loop, or the expression 'a+1' set
    // after the last wait in the loop
    if ( a > 4 ) break;
    wait();
    a++;
}
```

If the example above is changed to:

```
int a = 4;
wait();
while( 1 )
{
    if ( a > 4 ) break;
    a++;
    wait();
}
```

Agility can build it more efficiently as `a` is registered at each `wait` statement and the loop conditional is only combinationally dependent on the `>`.

Agility attempts to perform this transformation automatically. When finding a loop it tries to:

- Move any preceding `wait` statement so that it is immediately before the loop
- Move the last `wait` in a loop so that it is the last statement of the loop body.

It will only perform the transformation if it can make both moves, subject to the following restrictions.

- A `wait` statement cannot be moved past a statement that has an effect outside the current thread (`sc_signal`/`sc_port`/`sc_fifo` read or write)
- A `wait` statement cannot be moved past a loop
- A `wait` statement cannot be moved past an `if` statement.

This optimisation will never be applied to `do… while` loops, since the final statement in a normalised `do… while` is a conditional break.

Fine-grained logic sharing

Agility attempts to share logic within a thread if you have set ***resource constraints*** (see page 112). If you do not specify any resource constraints, no sharing takes place. You can provide constraints for:

- multipliers
- adders
- subtractors
- relational operators: less than (<); less than or equal to (<=); greater than (>); greater than or equal to (>=)

Agility can share operators when:

- it detects operations are in different branches of a conditionally executed block
- it detects operations are in different clock cycles

For each process, Agility reports:

- the total number of operations of each type
- the theoretical minimum number of resources of each type needed assuming no special cases
- the actual number of resources needed once sharing has been completed.
- the number of special cases found

In ***special cases*** (see page 123) it may not always be possible for Agility to meet the minimum number of resources. If Agility cannot meet the resource constraints set by the user, it will automatically add the necessary resources and inform the user.

Example

```
a = b*c;
wait();
d = e*f;
wait();

if (Condition)
    a = b*c;
else
    d = e*f;
```

In both cases above, only one multiplier will be built and shared between the two locations in the code.

Special case

It is not always possible to share entire expressions to produce the minimum hardware. To share two expressions fully, the order of evaluation of the operations in each expression must be the same. For example:

```
a = (b * c) + d;    // multiply followed by add
wait();
e = f * (g + h);    // add followed by multiply
```

In the expressions on the right-hand side, the order of evaluation of the operations is different.

If a user placed a constraint of 1 multiplier and 1 adder on this code, Agility could not satisfy these constraints.

Instead, Agility would automatically add an extra resource for an operation. Agility will always choose to add any resources in such a way as to minimize the increase in logic size. In the example above, Agility will choose to create 1 multiplier and 2 adders.

Agility optimizations

When Agility has performed high-level synthesis, it can then proceed to optimize the resulting circuit, for either speed or area.

Optimizations performed are:

High-level optimizations:

- Common sub-expression elimination
- Rewriting
- Conditional rewriting

Speed/area optimizations:

- Retiming
- Logic sharing
- Automatic tree balancing

Common sub-expression elimination

Common sub-expression elimination (CSE) identifies any logic that performs the same function from the same set of inputs and with the same type of output. CSE removes the copies and wires the inputs and outputs to the first copy.

Example

```
if (a == 0)
    b++;
wait();
c = (a == 0);
```

Only one copy of the comparison of `a` with 0 will be built and wired to where the result is needed. The user does not therefore have to try to explicitly re-use a single piece of code that performs the same function as elsewhere.

```
while(1)
{
    a++;
    wait();
    while (a != 10);
    {
        a++;
        b = ~b;
        wait();
    }
    a++;
    wait()
}
```

You may assume that three copies of `a++` will be built, but CSE will see that it is being asked to do the same thing on the same register three times. Therefore only one copy of `a++` will be built and the enable OR-ed between three control states.

Rewriting

Rewriting removes gates that always have the same output value and reduces the number of inputs to gates where possible. For example, in the statement:

```
if (a || ~a) // if (1)
```

`(a || ~a)` can be simplified to 1.

Conditional rewriting

Conditional rewriting (CR) applies automatically generated test patterns to netlists. It discovers if a gate (or input) is ever used, and removes it if not. For example:

```
if (a && (b || a))
```

CR removes `b` from the expression as it has no effect on the outcome of the evaluation of: `a && ( b || a ).`

With rewriting and conditional rewriting, the user is not required to simplify their Boolean equations to the simplest form possible, as the tool will do so automatically.

Automatic tree balancing

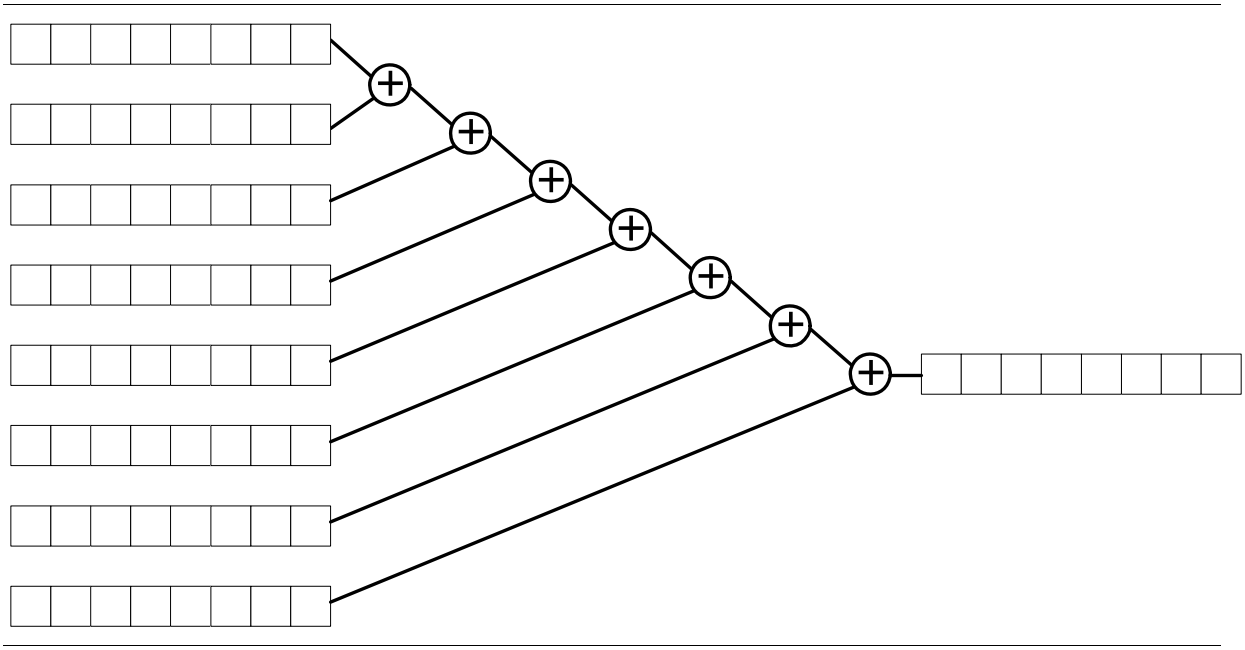
Agility can take a tree of operations, such as adds, and rebalance them to create shallower logic.

Without automatic tree balancing, the user needs to modify the input code such that it operates on pairs of inputs in rounds until the final result is produced. This process is non-trivial and involves modification from the original algorithmic source code. Automatic tree balancing produces a balanced tree without the need to modify the input source.

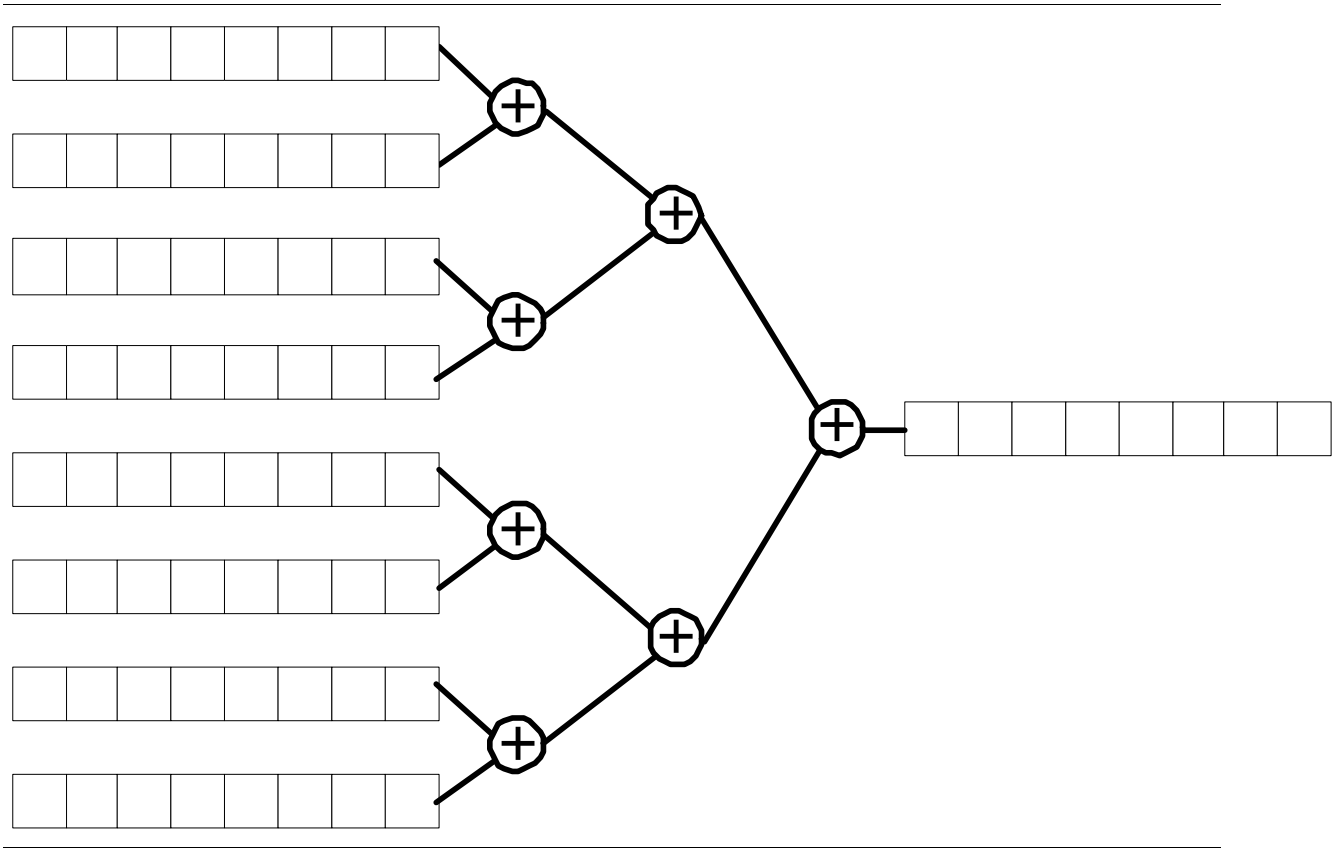
Example

```
char Array[8];
char Result;

wait();
Result = Array[0];
for (int i = 1; i<8 ; i++)
{
    Result = Result+Array[i];
}
wait();
```

Without automatic tree balancing, this would create an unbalanced tree of adds like this:



With automatic tree balancing, this balanced tree of adds is created:



Because tree-balancing may re-order operations, you may wish to temporarily switch off tree-balancing to ensure that the line number in an error message matches that in the original source code.

Technology mapping

The Agility technology mapper maps general logic into device-specific logic blocks.

Technology mapping is available for EDIF output for the following devices:

- Actel: ProASIC and ProASIC+
- Altera: Apex 20K, 20KE and 20KC, Apex II, Excalibur, Cyclone, Cyclone II, Stratix, Stratix GX, Stratix II
- Xilinx: Virtex, VirtexE, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, Spartan-II, Spartan-IIE, Spartan-3/3E/3L

Creating technology mapped EDIF

- Tick the Enable technology mapper option on the Synthesis node in Project Settings.

OR

- Use the `--technology-mapping` option in the command-line compiler

Technology mapping is off by default.

The SystemC compiler can generate an estimate of the number of look-up tables and other resources that will be used by the technology-mapped EDIF, using the logic estimator.

Optimizing arithmetic hardware in Actel devices

If you are targeting Actel ProASIC or ProASIC+ devices, you can optimize arithmetic hardware for area or for speed in EDIF output from your SystemC code.

In the Agility IDE, open the Synthesis node in Project Settings (you need to select a file in the left pane to see this tab). In the Expand netlist for: box, choose Area (to minimize area) or Speed (to maximize speed). The default setting is Speed.

If you are using the command-line compiler, use the option:

`--netlist-opt=area`	minimizes area
`--netlist-opt=speed`	maximizes speed

The area and speed settings affect adders, subtractors, multipliers and dividers in Actel devices. They have no effect for Altera and Xilinx devices.

ALU mapping

Some FPGA devices possess embedded ALU primitives, which the compiler has the ability to target automatically. Rather than leave it up to the user to specify where special ALU units should be used, the compiler intelligently uses them where they will provide the most improvement in performance over the equivalent logic.

Enabling mapping to ALUs

To turn ALU mapping on, use the Enable mapping to ALUs box on the Synthesis node in `Project Settings`.

If you are using the command-line compiler, use the `alu-mapping` option:

`--alu-mapping 1`	enable ALU mapping (default)
`--alu-mapping 0`	disable ALU mapping

Limiting the number of mapped ALUs

You can set the maximum number of ALUs of a specific type that the compiler targets. This is useful if not all ALUs on the device are available for a design. To limit the number of mapped ALUs, choose an ALU type from the Limit ALUs of type: box and enter the maximum number that the compiler can target.

From the command line, use the `--alu-limit` ***ALUType*=*Limit*** option.

E.g. `agility --edif -f XilinxVirtexII -p xc2v2000bg456-5 --alu-limit MULT18x18=100`

Supported ALU primitives

Family	*ALU resource*	*Supported Configurations*
Xilinx Virtex-II	MULT18X18	Simple multiplier
Xilinx Virtex-II Pro	MULT18X18	Simple multiplier
Xilinx Virtex-II Pro X	MULT18X18	Simple multiplier
Xilinx Virtex-4	DSP48	Simple multiplier
Xilinx Spartan 3	MULT18X18	Simple multiplier
Altera Cyclone II	CYCLONEII_DSP	Simple multiplier
Altera Stratix	STRATIX_DSP	Simple multiplier
Altera Stratix GX	STRATIX_DSP	Simple multiplier
Altera Stratix II	STRATIXII_DSP	Simple multiplier

Retiming

Retiming moves registers through the logic to minimize the logic depth in order to meet timing constraints. In Agility this feature allows the programmer to express pipelines that will be balanced automatically.

The retiming option moves flip-flops around in the circuit to try to meet the clock period specified on the clock (using ag_constrain_clock) for a given thread. It preserves:

- timing for logic between clock domains
- timing between flip-flops and external interfaces.

It moves:

- flip-flops in other parts of the circuit until the respective clock periods are met.
- flip-flops are then moved around again to minimize their number in the circuit, whilst conserving the specified clock periods.

Retiming is currently only available for Altera and Xilinx devices which are supported by the Technology Mapper. Retiming is enabled by default for EDIF output for Xilinx devices, but not for Altera devices.

Limitations

Although having insufficient pipeline registers is the most common issue constraining the upper speed of a design, there are also a number of other restrictions that apply to retiming moves that can affect the final clock rate of a design.

- Flips-flops on every input or output

  There must be a flip-flop on every input of a gate for a forward move, or every output of a gate for a backward move.
- Flip-flops all of the same class (share reset, clock and clock-enable wires)
- Flip-flops for initialized or uninitialized variables

  If a register is declared as initialized, then when the flip-flops it is composed of are moved during retiming the resultant circuit must have the same initial state.

5.1 Agility output formats

Agility supports the following output formats:

- ***EDIF*** (see page 130)
- ***VHDL*** (see page 132)
- ***Verilog*** (see page 135)
- ***RTL SystemC*** (see page 139)

The type of output produced is specified in the active configuration in the IDE, or as an option to the command-line compiler.

The output can be tailored for the tool used for simulation or place and route.

5.1.1 EDIF

The EDIF produced by Agility is EDIF 2 0 0, and conforms to the American National Standard (ANSI/EIA-548-1988, approved: March 14, 1988).

Block names

Instance names of cells in EDIF (blocks) have the form:

`B` ***blockId_fileName_lineNumber_moduleInfo_blockType***

where:

`B`	Indicates that the current name is for a block (as opposed to a net).
blockId	The unique Id for the block within its name scope.
fileName	The name of the file containing the source code from which the block was generated. It forms part of the name scope for the block.
lineNumber	The line number in the source code from which the block was generated.
moduleInfo	Extra information about the SystemC module instance that this block originates from. If the block originates from a module that is instantiated in another module, the instance names of these modules are combined.
blockType	A string identifying the type of block in question (i.e. whether it is a register, an AND gate, a pad, etc).

Examples

```
B2_filea_sc_cpp_55_Top_Instance0_DTYPE0
B50_fileb_sc_cpp_10_Top_Instance0_Instance_2_AND
```

Net names

Net names in the EDIF output are either assigned a name or take their name from the block that feeds them.

The former are referred to as named nets and are assigned a specific name often corresponding to a variable or signal in SystemC. For example, suppose SystemC declares the following register variable:

```
sc_uint <8> MyVar;
```

then the EDIF will contain 8 nets named `MyVar_Reg_0` to `MyVar_Reg_7`.

There are other named nets that are generated internally by the Agility and do not refer back to constructs in the SystemC source, for example `ControlWireId_1596`.

If a net is not associated with a name, it is referred to as unnamed and it will take its name from the block that feeds it:

`W [G][T]`***blockId_filename_lineNumber_moduleInfo_port***

where:

`W`	Indicates that the current name is for a net (as opposed to a block).
`G`	Optional. Indicates that a net is global and crosses file or function boundaries.
`T`	Optional. Indicates that a net is global and crosses file or function boundaries.
blockId	The unique Id of the block feeding this net
fileName	The name of the file containing the source code from which the block that feeds this net was generated.
lineNumber	The line number in the source code from which the block that feeds this net was generated.
moduleInfo	Extra information about the SystemC module instance where thee block that feeds this net originates from.
blockType	A string identifying the type of block that feeds this net.
port	Name of the output port to which this net connects to the block feeding this net.

Examples

```
W2_filea_sc_cpp_55_Top_Instance0_DTYPE0_Q
WG50_fileb_sc_cpp_10_Top_Instance0_Instance_2_AND_O
```

If two named nets take the same name, they will be output as if they are unnamed to ensure that they have unique identifiers.

Port names

The names of top-level ports in SystemC are preserved in the EDIF output. For example:

```
class MyModule : public sc_module
{
public:
  sc_in<bool> Clock;
  sc_in< sc_uint<2> > MyInput;
  sc_out< sc_uint<2> > MyOutput;

  ...
}
```

produces three ports in EDIF: `Clock`, `MyInput` and `MyOutput`. If the option `--ports-as-pins` is set, Agility treats the ports as single-bit pins and the EDIF will have five ports: `Clock`, `MyInput_0`, `MyInput_1`, `MyOutput_0` and `MyOutput_1`.

5.1.2 VHDL

Agility produces VHDL that is compliant with IEEE 1076.6 standard for RTL synthesis. It is hierarchically structured and reflects the structure of the SystemC input.

VHDL entity naming

Each SystemC module (i.e. class derived from `sc_module`) instantiation creates a separate entity in VHDL rather than an instance of the same entity. This allows modules in SystemC to be parameterized. The name of the entity is taken from the module name in SystemC, which may be passed to the constructor of `sc_module`. For example, if there is a class called `MyModule`:

```
class MyModule : public sc_module
{
public:
  MyModule( sc_module_name name ) : sc_module( name )
  {
     ...
  }
}

void ag_main()
{
  MyModule TheModule( "Top" )
}
```

then the top-level entity in VHDL will be named `Top`. If no name is passed to the constructor of `sc_module`, the name of the entity will be taken from the type name, for example:

```
class MyModule : public sc_module
{
public:
  MyModule() : sc_module()
  {
     ...
  }
}

void ag_main()
{
  MyModule TheModule( )
}
```

will produce an entity called `MyModule`.

VHDL hierarchy

If a module of type `MyModuleA` instantiates another module of type `MyModuleB`, the corresponding names are combined. For example:

```
class MyModuleA : public sc_module
{
public:
  MyModuleA( sc_module_name name ) : sc_module( name )
  {
     ...
  }
}

class MyModuleB : public sc_module
{
public:
  MyModuleA * subModule;

  MyModuleB( sc_module_name name ) : sc_module( name )
  {
     subModule = new MyModuleA( "Instance" );
  }
}

void ag_main()
{
  MyModule TheModule( "Top" )
}
```

results in two VHDL entities: `Top` and `Top_Instance`.

If more than one module instantiation would produce the same name in VHDL, the first instance will be named as described above. Subsequent instantiations will have numbers appended to their names, starting at 1, for example:

```
class MyModuleA : public sc_module
{
public:
  MyModuleA() : sc_module()
  {
     ...
  }

  MyModuleA( sc_module_name name ) : sc_module( name )
  {
     ...
  }
}

class MyModuleB : public sc_module
{
public:
  MyModuleA * subModule1, * subModule2, * subModule3;

  MyModuleB( sc_module_name name ) : sc_module( name )
  {
     subModule1 = new MyModuleA( "Instance" );
     subModule2 = new MyModuleA( "Instance" );
     subModule3 = new MyModuleA();
  }
}

void ag_main()
{
  MyModule TheModule( "Top" )
}
```

results in four entities, named `Top`, `Top_Instance`, `Top_Instance_1` and `Top_MyModuleA`.

It is also possible to generate non-hierarchical VHDL where Agility produces just one entity for the whole design. Use the `--no-hierarchy` option to do so.

VHDL file names

Each VHDL entity produced by Agility will be output to a separate file. This name of this file has the same name as the entity, except for the file containing the top-level entity, the name of which is specified with the `--output-file` ***flag*** (see page 184).

VHDL threads

To make it easier to relate the VHDL generated by Agility back to the input SystemC, each VHDL module is divided into sections. Each section corresponds to a thread in the SystemC and the sections are annotated with the name of the corresponding thread.

VHDL port and signal names

The names of top-level ports in SystemC are preserved in the top-level entity of the VHDL output. For example:

```
class MyModule : public sc_module
{
public:
  sc_in<bool> Clock;
  sc_in< sc_uint<2> > MyInput;
  sc_out< sc_uint<2> > MyOutput;

  ...
}
```

results in three ports in VHDL: `Clock`, `MyInput` and `MyOutput`. If the option `--ports-as-pins` is set, Agility treats the ports as single-bit pins and the VHDL will have five ports: `Clock`, `MyInput_0`, `MyInput_1`, `MyOutput_0` and `MyOutput_1`.

Signal names in VHDL have names that are either user-defined or compiler-generated, depending on their origin. Signals originating from user-defined logic, such as outputs of data registers, have names related to the source SystemC. For example, if the SystemC has a register variable called `MyVar`, the output from the corresponding register in VHDL is called `MyVar_Reg`. This should make it easier to relate the produced VHDL back to the SystemC source.

VHDL port types

Ports in Agility-produced VHDL are of type `std_logic` for single-bit and `numeric_std.unsigned` for other ports by default.

To change the type of a port in VHDL, use the `ag_constrain_port` (see page 114) routine.

5.1.3 Verilog

Agility produces Verilog that is compliant with both the IEEE `1364-1995` (default) and the IEEE `1364-2001` standards.

Targeting Verilog-2001

When targeting Verilog-2001, Agility will output the following additional constructs:

- Combined port and data type declarations
- Module port parameter lists
- Comma separated sensitivity lists
- Combinational logic sensitivity lists using a wildcard
- Signed arithmetics
- Initialization of variables at declaration

The Verilog that is produced is hierarchically structured and reflects the structure of the input SystemC.

Verilog module naming

Each SystemC module (i.e. class derived from `sc_module`) instantiation creates a separate module in Verilog rather than an instance of the same module, which is done to cope with parameterized modules in SystemC. The name of the module in Verilog is taken from the module name in SystemC, which may be passed to the constructor of `sc_module`. For example, if there is a class called `MyModule` as follows:

```
class MyModule : public sc_module
{
public:
  MyModule( sc_module_name name ) : sc_module( name )
  {
     ...
  }
}

void ag_main()
{
  MyModule TheModule( "Top" )
}
```

The top-level module in Verilog will be named `Top`. If no name is passed to the constructor of `sc_module`, the name of the module will be taken from the type name, for example:

```
class MyModule : public sc_module
{
public:
  MyModule() : sc_module()
  {
     ...
  }
}

void ag_main()
{
  MyModule TheModule( )
}
```

will produce a module called `MyModule`.

Verilog hierarchy

If a module of type `MyModuleA` instantiates another module of type `MyModuleB`, then the corresponding names as described in the previous section are combined, for example:

```
class MyModuleA : public sc_module
{
public:
  MyModuleA( sc_module_name name ) : sc_module( name )
  {
     …
  }
}

class MyModuleB : public sc_module
{
public:
  MyModuleA *subModule;

  MyModuleB( sc_module_name name ) : sc_module( name )
  {
     subModule = new MyModuleA( "Instance" );
  }
}

void ag_main()
{
  MyModule TheModule( "Top" )
}
```

will produce two Verilog modules: `Top` and `Top_Instance`.

If more than one module instantiation would produce the same name in Verilog, the first instance will be named as described above. Subsequent instantiations will have numbers appended to their names, starting at 1, for example:

```
class MyModuleA : public sc_module
{
public:
  MyModuleA() : sc_module()
  {
     …
  }

  MyModuleA( sc_module_name name ) : sc_module( name )
  {
     …
  }
}

class MyModuleB : public sc_module
{
public:
  MyModuleA *subModule1, *subModule2, *subModule3;

  MyModuleB( sc_module_name name ) : sc_module( name )
  {
     subModule1 = new MyModuleA( "Instance" );
     subModule2 = new MyModuleA( "Instance" );
     subModule3 = new MyModuleA();
  }
}

void ag_main()
{
  MyModule TheModule( "Top" )
}
```

will produce four modules: `Top`, `Top_Instance`, `Top_Instance_1` and `Top_MyModuleA`.

It is also possible to generate non-hierarchical Verilog, where Agility produces just one module for the whole design. Use the `--no-hierarchy` option to do so.

Verilog file names

Each Verilog module produced by Agility is output to a separate file. This file has the same name as the module, except for the file containing the top-level module, the name of which is specified with the `--output-file` ***flag*** (see page 184).

Verilog threads

To make it easier to relate the Verilog generated by Agility back to the input SystemC, each Verilog module is divided into sections. Each section corresponds to a thread in the SystemC and the sections are annotated with the name of the corresponding thread.

Verilog port and wire names

The names of top-level ports in SystemC are preserved in the top-level entity of the Verilog output. For example:

```
class MyModule : public sc_module
{
public:
  sc_in<bool> Clock;
  sc_in< sc_uint<2> > MyInput;
  sc_out< sc_uint<2> > MyOutput;

  ...
}
```

results in three ports in Verilog: `Clock`, `MyInput` and `MyOutput`. If the option `--ports-as-pins` is set, Agility treats the ports as single-bit pins and the Verilog will have five ports: `Clock`, `MyInput_0`, `MyInput_1`, `MyOutput_0` and `MyOutput_1`.

Wire names in Verilog have names that are either user-defined or compiler-generated, depending on their origin. Wires originating from user-defined logic, such as outputs of data registers, have names related to the source SystemC. For example, if the SystemC has a register variable called `MyVar`, the output from the corresponding register in Verilog is called `MyVar_Reg`. This should make it easier to relate the produced Verilog back to the SystemC source.

5.1.4 RTL SystemC

The RTL SystemC Agility produces is optimized for simulation speed so it has a different structure to the original SystemC input.

RTL SystemC module names

Each SystemC source file produces an RTL SystemC module, containing all logic defined in that file. The name of the module is taken from the name of the file. If several files with the same name are compiled, the output SystemC will have one module with the name of these files, while subsequent modules have a number appended to them, starting with 1.

A top-level module is produced as well, which instantiates all other produced modules and connects them together. The name of this module is specified using the `--output-file` (see page 184) option.

It is also possible to generate non-hierarchical RTL SystemC, where Agility produces just one module for the whole design. Use the `--no-hierarchy` option to do so.

RTL SystemC file names

Each RTL SystemC module produced by Agility is output to a separate file. This file has the same name as the module.

RTL SystemC port and variable names

The names of top-level ports in SystemC are preserved in the top-level module of the RTL SystemC output. For example:

```
class MyModule : public sc_module
{
public:
  sc_in<bool> Clock;
  sc_in< sc_uint<2> > MyInput;
  sc_out< sc_uint<2> > MyOutput;

  …
}
```

produces three ports: `Clock`, `MyInput` and `MyOutput all having the same type as the original SystemC ports`. If the option `--ports-as-pins` is set, Agility treats the ports as single-bit pins and the produced SystemC will have five ports: `Clock`, `MyInput_0`, `MyInput_1`, `MyOutput_0` and `MyOutput_1`, all of basic port type `bool`.

Variable and signal names in RTL SystemC have names that are either user-defined or compiler-generated, depending on their origin. Variables and signals originating from user-defined logic, such as outputs of data registers, have names related to the source SystemC. For example, if the SystemC has a register variable called `MyVar`, the output from the corresponding register in the RTL SystemC is called `MyVar_Reg`. This should make it easier to relate the produced RTL SystemC back to the SystemC source.

5.1.5 HDL output formats

Agility can customize the HDL output for particular third-party tools.

Synplify and Precision HDL
The Synplify and Precision output styles are syntactically identical. The differences between them are synthesis directives, giving variations in the format of:

- pin location constraints
- black box attributes
- net preservation attributes
- RAM initializers when an FPGA family is specified (when using RAMs)

These are not really differences in the HDL, but only in the tool dependent synthesis directives.

ModelSim and Active-HDL
ModelSim and Active-HDL output styles are identical. When specifying one of these styles:

- All signals are initialized such that they have a value on start-up when simulating.
- Memories are output as a generic model, rather than a device specific primitive.
- The VHDL does contain any synthesis directives.

Otherwise, the HDL is identical to the Synplify and Precision output styles.

Generic
The generic output style is very similar to the ModelSim/ActiveHDL output styles, but signals are not initialized on start-up because this style is meant for synthesis.

5.1.6 HDL output references

- VHDL for logic synthesis
  Author: Andrew Rushton
  Publisher: John Wiley and Sons
  ISBN: 0-471-98325-X
  Published: May 1998

- IEEE standard 1364 -1995
  IEEE Standard Hardware Description Language
  Based on the Verilog® Hardware Description Language.
  `http://standards.ieee.org/`

- EDIF 2 0 0 American National Standard
  (ANSI/EIA-548-1988, approved: March 14, 1988).

5.2 Facilities for reviewing synthesis results

Agility can generate the following reports and graphs to enable you to review your results.

Celoxica CDFG viewer (see page 166)

Logic estimator

Report files

5.2.1 Logic estimator

The SystemC compiler can give information on logic usage and depth to help you optimize your designs. (Note that this information is based on estimates, since full place and route is needed to get exact logic and area information.)

Logic estimation information is only available for EDIF builds.

Using the logic estimator
To generate information about the logic area and depth of your code:

- Set the Generate estimation info box to True in the Linker>Output node of the Project Settings dialog.

  OR

- Use the `--estimate-logic-size` option on the command line compiler. For example:
  `agility --estimate-logic-size=0 test.sc.cpp`

The information generated is most detailed for builds targeting devices supported by the Technology Mapper (with the technology mapper enabled).

When you compile your code a set of HTML files will be produced, containing:

- Line by line information on use of resources (e.g. NAND gates, or look-up tables for mapped-EDIF).
- Description of the longest combinational paths in your code.

You can access the information by opening the overview page `summary.html` in the Agility IDE. `summary.html` will be placed in the build directory for your project.

Logic area and depth summary

You can view logic area and depth information about your code by opening the file `summary.html`

The file is created in your build directory by the logic estimator if you have selected the Generate estimation info option.

Area estimation information

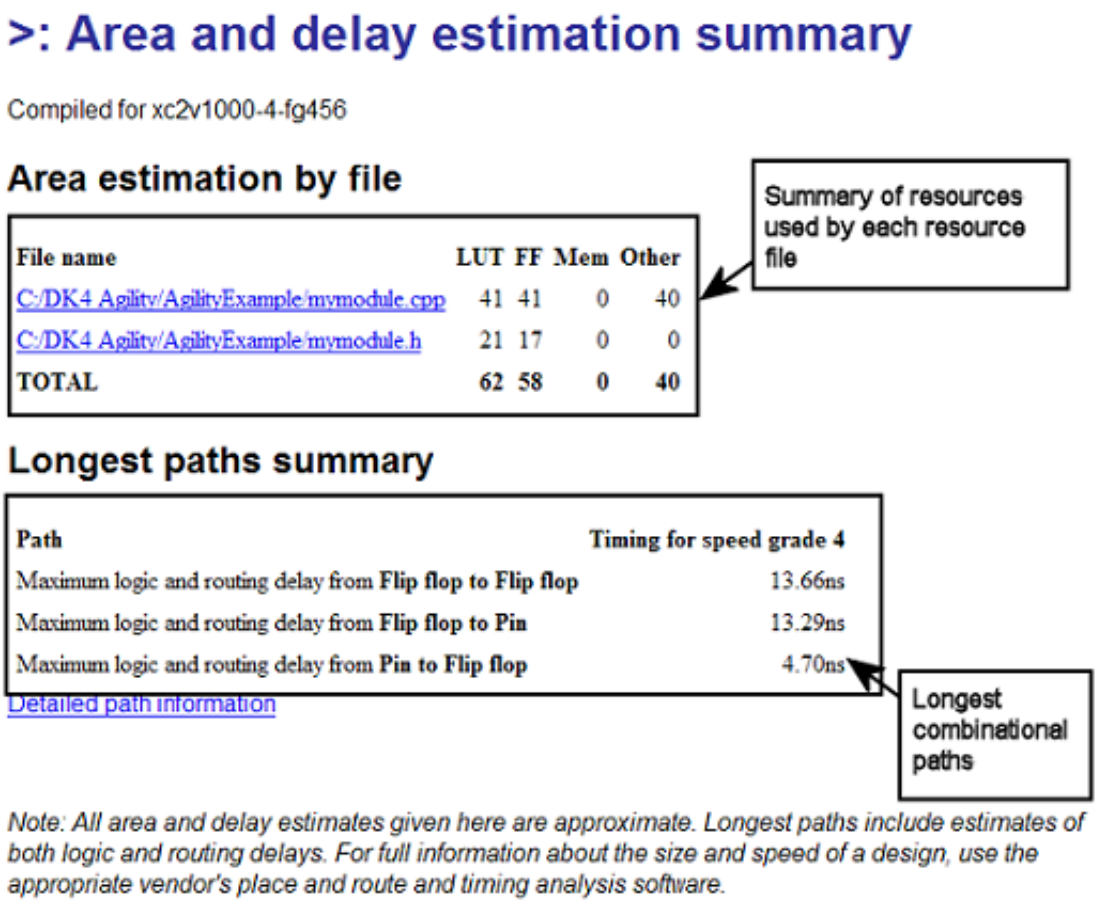
- For technology mapped-EDIF: consists of the number of look-up tables (LUTs), flip-flops (FFs), memory resources and other device-specific logic resources (listed under "Other").
- For non-mapped EDIF: consists of the number of NAND gates, flip-flops and memory resources.

Each source file listed is linked to more detailed logic area information.

Longest paths summary

`summary.html` displays estimates of the longest path for each combination of flip-flops to/from pins, RAMs and flip-flops, pins to/from RAMs and pins, RAMs to RAMs. It also links to more detailed combinational path information.

Example: Area and delay estimation


>: Area and delay estimation summary
Compiled for xc2v1000-4-fg456
Area estimation by file
File name LUT FF Mem Other
C:/DK4 Agility/AgilityExample/mymodule.cpp 41 41 0 40
C:/DK4 Agility/AgilityExample/mymodule.h 21 17 0 0
TOTAL 62 58 0 40
Summary of resources used by each resource file
Longest paths summary
Path Timing for speed grade 4
Maximum logic and routing delay from Flip flop to Flip flop 13.66ns
Maximum logic and routing delay from Flip flop to Pin 13.29ns
Maximum logic and routing delay from Pin to Flip flop 4.70ns
Detailed path information
Longest combinational paths
Note: All area and delay estimates given here are approximate. Longest paths include estimates of both logic and routing delays. For full information about the size and speed of a design, use the appropriate vendor's place and route and timing analysis software.

Information on logic area

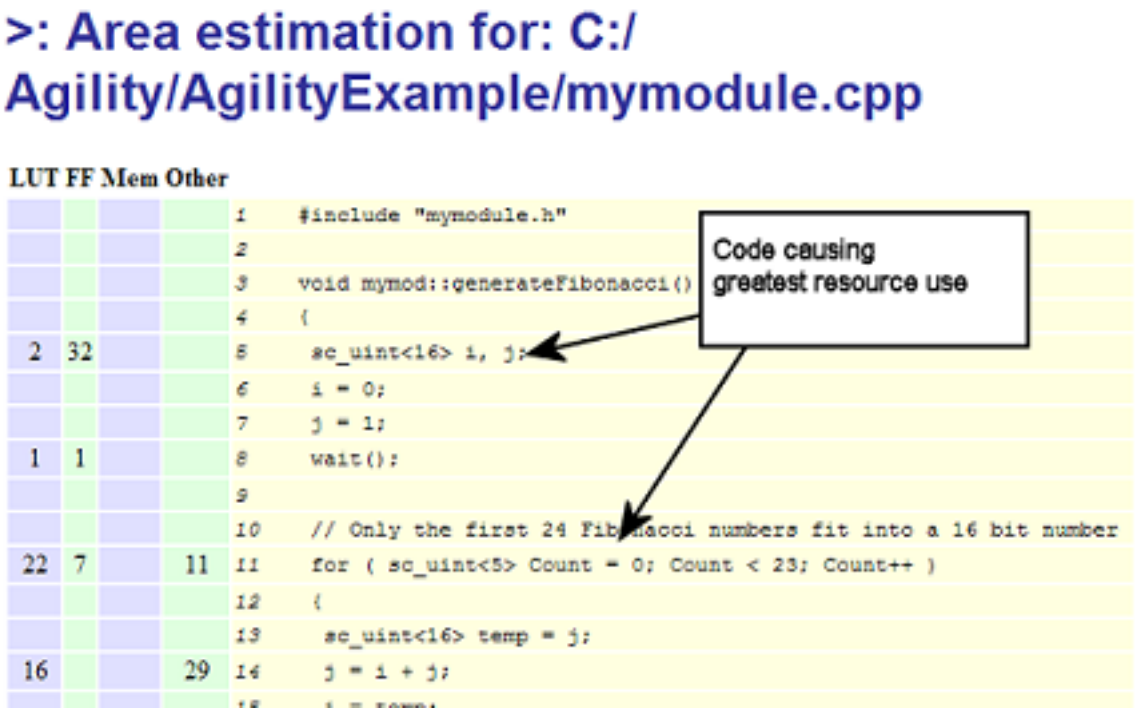
The detailed information about area provided by the logic estimator consists of the number of resources created for each line of your source code. Totals are summarized in the overview page, `summary.html`.

For each line of code, the areas that use the greatest resource in that line are highlighted in colour. Red code provides 75% or more of the maximum, orange code 50 -75% of the maximum, and blue 25 - 50%. Black code contributes up to 25% of the maximum.

The number of resources used is listed next to each line of code. Resources listed are:

- LUT: look-up tables (mapped-EDIF output only)
- NAND: NAND gates (non-mapped EDIF only)
- FF: Flip flops
- Mem: Memory bits
- Other: device-specific logic elements (mapped-EDIF output only):
  - Altera: CARRY_SUM, CARRY
  - Xilinx: MuxF5, MuxF6, MuxF7, MuxF8, MuxCY, XORCY, MultAND

Example: Logic area estimation


>: Area estimation for: C:/ Agility/AgilityExample/mymodule.cpp
LUT FF Mem Other
1 #include "mymodule.h"
2
3 void mymod::generateFibonacci()
4 {
2 32 5 sc_uint<16> i, j;
6 i = 0;
7 j = 1;
1 1 8 wait();
9
10 // Only the first 24 Fibonacci numbers fit into a 16 bit number
22 7 11 11 for (sc_uint<5> Count = 0; Count < 23; Count++)
12 {
13 sc_uint<16> temp = j;
16 29 14 j = i + j;
Code causing greatest resource use

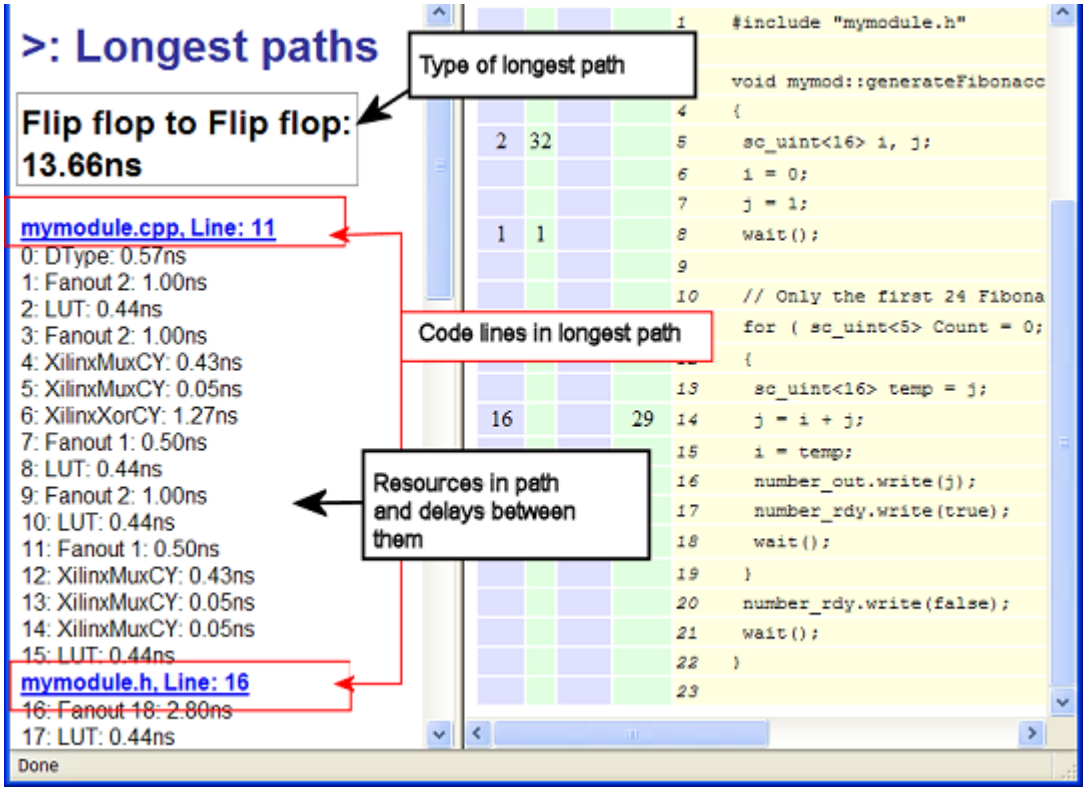
Information on combinatorial paths and delay

Information on logic delay generated by the logic estimator is summarized on the overview page, `summary.html`. This contains a link to more detailed information, where the longest combinational path is given for the following 9 path types. If that path does not exist, it is not included:

	From flip-flops	*From pins*	*From RAM*
To flip-flops	•	•	•
To pins	•	•	•
To RAM	•	•	•

For each of the longest paths, there is a list of the lines of source code that contribute to the path, and a list of resources used by each of these lines.

Example: Logic depth estimation



5.2.2 Report files

Agility generates a compilation report file in XML. The report includes:

- warnings and errors
- summary of hardware used
- area estimation
- block counts

- unused declarations
- optimization information, e.g. removed flip-flops and memories
- registers that could not be moved by the retimer

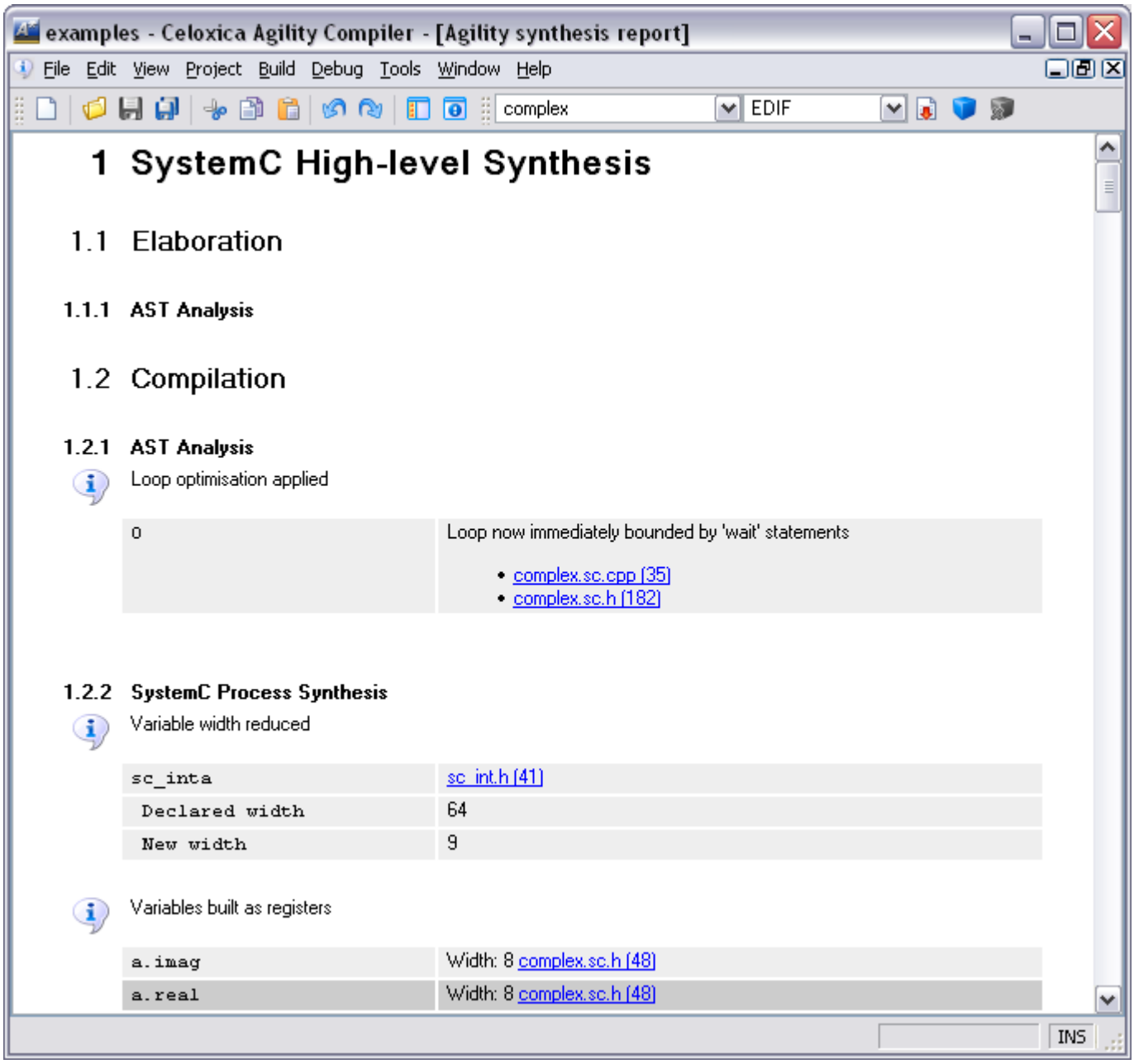
They are viewed within the IDE. To view a report, click on the message about the report. You may also use the links in the report to open the source files within the IDE.

Contents of the report

All the messages that appear during a compilation or build are added to the report.

Errors and warnings

The errors and warnings that appear in the Agility IDE also appear in the report.



Optimizations

The report gives details of optimizations performed by the compiler, such as reducing the size of a circuit by rearranging and removing components.

5.3 Preparing code for simulation

5.3.1 Simulating clocks correctly

The RTL SystemC produced by Agility contains a call to function `check_clock_start`, which checks if a clock edge occurs at time zero during simulation. If it does, `check_clock_start` generates a warning as methods that are sensitive to the clock may not have initialized before the circuit starts and therefore may not simulate correctly.

To solve this problem, the clock start needs to be delayed. To do this, create a clock using one of the following overloads of `sc_clock` specifying a positive start time:

```
  sc_clock (sc_module_name name_, const sc_time &period_, double
duty_cycle_=0.5, const sc_time &start_time_=SC_ZERO_TIME, bool
posedge_first_=true)
  sc_clock (sc_module_name name_, double period_v_, sc_time_unit
period_tu_, double duty_cycle_, double start_time_v_, sc_time_unit
start_time_tu_, bool posedge_first_=true)
  sc_clock (sc_module_name name_, double period_, double duty_cycle_=0.5,
double start_time_=0.0, bool posedge_first_=true)

sc_clock clk( "my_clock", 10, SC_NS, 0.5. 1.0 /* first clock edge */,
SC_NS, true );
```

For more information see the ***SystemC Language Reference Manual*** http://www.systemc.org.

5.4 Targeting hardware

5.4.1 Preparing to target hardware using Agility

You need to prepare your simulated code by converting the interfaces to hardware.

Before targeting hardware, you need to check that you have removed all debug statements from your code.

You need to select whether top-level ports should be output as pins or ports.

5.4.2 Converting interfaces to hardware

Once your program has been simulated correctly you must add the necessary hardware interfaces.

- Convert any file reading and writing procedures into interface or bus procedures.
- Convert any interfaces to simulations of hardware into interfaces to black box code or remove them entirely.

5.4.3 Top-level port synthesis

Agility can output the top-level ports either as pins in the EDIF and HDL output or as ports.

If the Agility design is to be included in a larger project, for example in a larger HDL design, they should be output as ports.

If the Agility design makes up the whole chip, then the top-level ports will need to be synthesized as pins.

This option can be set

In the IDE:

Go to Project settings, SystemC section and setting Infer Pins to true (to enable output of pins)

At the command line

using the `--ports-as-pins` option.

5.4.4 Targeting a particular synthesis tool

You need to specify an output style for VHDL or Verilog output. This enables the compiler to generate code that uses the features of the selected synthesis/simulation tool.

1. In the Project Settings dialog, ensure that the mode is VHDL or Verilog.
2. Select the Linker node.
3. In the Synthesis tool pull-down list, select the appropriate tool:
   - Aldec Active-HDL (used for simulation)
   - Altera QuartusII
   - Mentor Graphics ModelSim (used for simulation)
   - Mentor Graphics Precision
   - Simulation (generic simulation output)
   - Synplicity Synplify Pro
   - Synopsys Design Compiler
   - Synthesis (generic synthesis output)
   - Xilinx Synthesis Tool (XST)

Choose Synthesis if you want to use a synthesis tool which is not listed. Choose Active-HDL, ModelSim or Simulation if you want to simulate your code.

If you are using the command line compiler, use the `--hdl-type` ***SynthesisTool*** option.

5.4.5 Technology mapping

- Tick the Enable technology mapper option on the Synthesis node in Project Settings.

OR

- Use the `--technology-mapping` option in the command-line compiler

Technology mapping is off by default.

The SystemC compiler can generate an estimate of the number of look-up tables and other resources that will be used by the technology-mapped EDIF, using the logic estimator.

ALU mapping

Some FPGA devices possess embedded ALU primitives, which the compiler has the ability to target automatically. Rather than leave it up to the user to specify where special ALU units should be used, the compiler intelligently uses them where they will provide the most improvement in performance over the equivalent logic.

Enabling mapping to ALUs

To turn ALU mapping on, use the Enable mapping to ALUs box on the Synthesis node in `Project Settings`.

If you are using the command-line compiler, use the `alu-mapping` option:

`--alu-mapping 1` enable ALU mapping (default)
`--alu-mapping 0` disable ALU mapping

Limiting the number of mapped ALUs

You can set the maximum number of ALUs of a specific type that the compiler targets. This is useful if not all ALUs on the device are available for a design. To limit the number of mapped ALUs, choose an ALU type from the Limit ALUs of type: box and enter the maximum number that the compiler can target.

From the command line, use the `--alu-limit` ***ALUType*=*Limit*** option.

E.g. `agility --edif -f XilinxVirtexII -p xc2v2000bg456-5 --alu-limit MULT18x18=100`

Supported ALU primitives

Family	*ALU resource*	*Supported Configurations*
Xilinx Virtex-II	MULT18X18	Simple multiplier
Xilinx Virtex-II Pro	MULT18X18	Simple multiplier
Xilinx Virtex-II Pro X	MULT18X18	Simple multiplier
Xilinx Virtex-4	DSP48	Simple multiplier
Xilinx Spartan 3	MULT18X18	Simple multiplier
Altera Cyclone II	CYCLONEII_DSP	Simple multiplier
Altera Stratix	STRATIX_DSP	Simple multiplier
Altera Stratix GX	STRATIX_DSP	Simple multiplier
Altera Stratix II	STRATIXII_DSP	Simple multiplier

Optimizing arithmetic hardware in Actel devices

If you are targeting Actel ProASIC or ProASIC+ devices, you can optimize arithmetic hardware for area or for speed in EDIF output from your SystemC code.

In the Agility IDE, open the Synthesis node in Project Settings (you need to select a file in the left pane to see this tab). In the Expand netlist for: box, choose Area (to minimize area) or Speed (to maximize speed). The default setting is Speed.

If you are using the command-line compiler, use the option:

`--netlist-opt=area`	minimizes area
`--netlist-opt=speed`	maximizes speed

The area and speed settings affect adders, subtractors, multipliers and dividers in Actel devices. They have no effect for Altera and Xilinx devices.

5.4.6 Targeting hardware via EDIF

Set up your project to target EDIF using the Project Settings dialog. Set EDIF in the Output format setting in the Linker>Output node. This causes your project to compile directly to an `.edf` file which can be passed to your place and route tools.

Setting up place and route tools

The Altera EDIF compiler requires a library-mapping file. This is supplied as `celoxica.lmf`.

Preparing Quartus to compile SystemC EDIF

You need to set up Quartus in different ways depending whether you have compiled your EDIF using Agility, or used a synthesis tool to convert Agility VHDL or Verilog to EDIF.

Preparing Agility-generated EDIF, Quartus v. 5 (or later)

1. Start Quartus.
2. Create or open the project in which you want to compile the netlist generated by Agility.
3. Add the Agility-generated EDIF netlist, and any other design files, to the project.
4. Select the Assignments>EDA Tool Settings menu command.
5. In the EDA Tool Setting pane, select Design entry/synthesis as the Tool Type.
6. Select Custom as the Tool name from the drop-down list.
7. Set the Library Mapping File to specify the **`celoxica.lmf`** file installed in **`InstallDir\Agility\lmf`**.
8. Apply the TCL script that was generated by Agility when compiling the SystemC code to EDIF. The script file has the same file name as the compiled file.
9. To apply the script:
   - Enter the following command in the Quartus console window:
     `source cel_edif.tcl`
     where **`cel_edif`** is the name of the file compiled to EDIF.

   OR

   - Select the Tools>Tcl scripts.
   - Expand the Projects folder.
   - Select the TCL file to run and click Run.
     (The TCL files in the Projects folder will be those in the same directory as your EDIF files for the project).

You can now place and route your compiled design.

Trouble-shooting if Quartus cannot use a pin
If Quartus complains about being unable to use a pin (e.g. H19) then use the setting below in the Quartus GUI.

1. Select Device from the Assignments menu.
2. Click the Device & Pin options button.
3. Select the Dual purpose pins tab.
4. Choose Use as regular I/O for Data 0 (and/or possibly for the other options as well).

Agility HDL converted to EDIF using a synthesis tool
If you use Agility to generate VHDL or Verilog output, and then use a synthesis tool such as Synplify to compile this to EDIF, you need to select the Power-Up Don't Care option in Quartus :

If you are using the command line:

- In the Tcl console window type:
  `project add_assignment "" "" "" "" ALLOW_POWER_UP_DONT_CARE Off`
  Then press Return.

If you are using the IDE:

- For Quartus v. 5 (or later):
  Select Assignments>Analysis & Synthesis Settings>More Settings...
  Then choose Power-Up Don't Care from the Existing option settings: list, and set it to Off.

This only needs to be done once for the whole project. You do not need to set this option if you are compiling EDIF generated directly by Agility.

Importing timing constraint files into Actel Designer

To import `.gcf` files for Actel ProASIC and ProASIC+:

1. Start Designer.
2. Create or open the design in which you want to compile the netlist generated by Agility.
3. Import the Agility-generated EDIF netlist, and any other design files, to the project.
4. Use File>Import... to import the generated `.gcf` file that includes the timing constraints.
5. Compile the design.
6. Ensure that you select Timing driven for the layout.
7. Layout the design.

You can now place and route your compiled design.

5.4.7 Targeting hardware via Verilog

Set up your project to target Verilog using the Project Settings dialog.

- Set Verilog in the Output format setting in the Linker>Output node.
- Specify the synthesis tool that you are using the Linker tab in Project Settings. If you wish to simulate your Verilog, select ModelSim as the tool used.

Your project will then compile directly to a single `.v` file containing one module which can be passed to your synthesis or simulation tools. .

You can select which tool the produced Verilog is passed to by using the VHDL/Verilog style in the Linker>Output node of the Project Settings. You must include `celoxica.v` for synthesis/simulation, which can be found in ***InstallDir***`\Agility\verilog`**.**

Verilog 2001

Agility supports Verilog 2001 (IEEE 1364-2001).

From the Project Settings dialog:

Select the `Verilog 2001` Output format in the Linker > Output node.

From the command-line:

Use the `--verilog-2001` option.

When targeting Verilog 2001, Agility will output the following Verilog 2001 constructs:

- Combined port and data type declarations
- Module port parameter lists
- Comma separated sensitivity lists
- Combinational logic sensitivity lists using a wildcard
- Signed arithmetics
- Initialization of variables at declaration

5.4.8 Targeting hardware via VHDL

Set up your project to target VHDL using the Project Settings dialog.

- Set VHDL in the Output format setting in the Linker>Output node.
- Specify the synthesis tool that you are using in the Linker node in Project Settings. If you wish to simulate your VHDL, select ModelSim or Active-HDL as the tool used.

Your project will then compile directly to a single `.vhd` file which can be passed to your synthesis or simulation tools. You must include the file `celoxica.vhd` which can be found in ***InstallDir***`/<extpath>/vhdl`.

Module names are related to entity names in the generated VHDL code

6 Using Agility

You can use the Agility Compiler

- from within the Integrated Development Environment (IDE), which provides a graphical interface to help you build your projects and a code editor to maintain your program source files.
- on a command-line, either directly in a shell, or indirectly through a third-party build tool such as GNU make.

This section covers instructions for using:

- The IDE
- The command line interface to the Agility Compiler.

7 Agility IDE guide

7.1 Starting the IDE

Start the Integrated Development Environment by:

- Using the Agility icon on the desktop (MS Windows) .
- Using the Agility shortcut in the Celoxica > Agility program group of the Start menu (MS Windows).
- Executing the command celdev (Linux).

7.2 IDE overview

The IDE is a Multiple Document Interface (MDI) style GUI application with three distinct regions.

- Workspace window - a docking window (to the right of the main window frame by default).
- Output window - a docking window (at the bottom of the main window frame by default).
- MDI area - the central region in which code editor windows are shown.

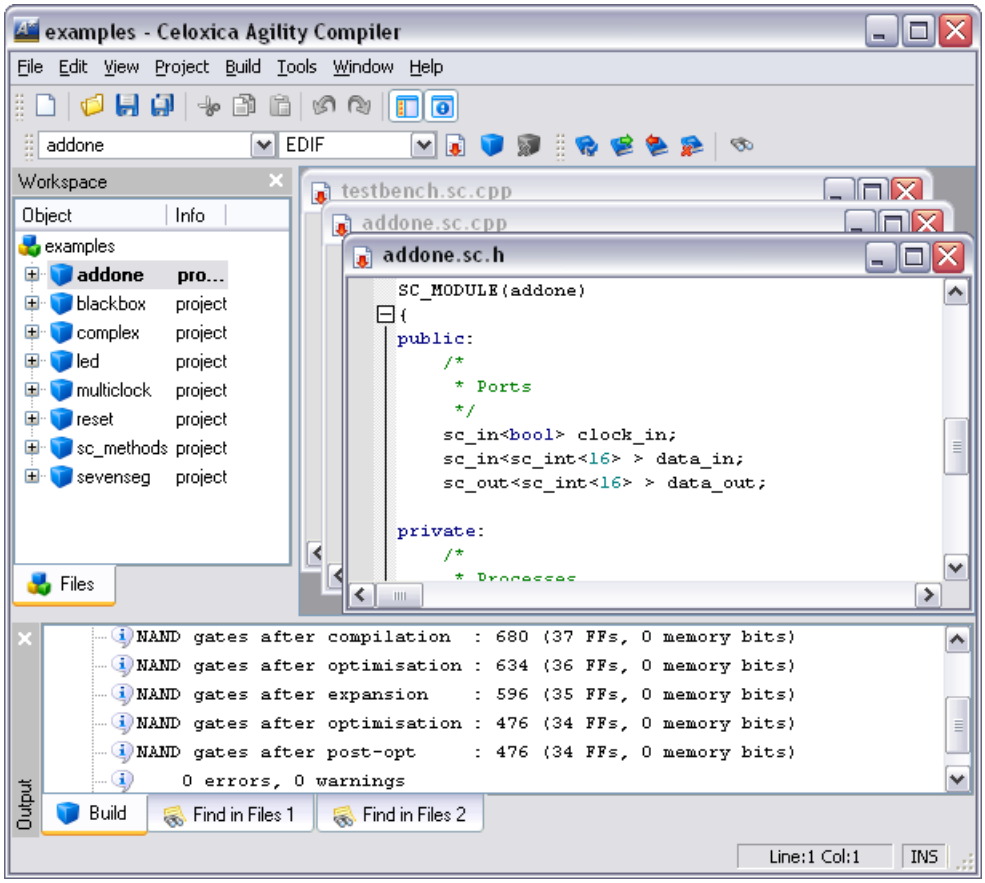


Figure 1: IDE tri-pane window

7.2.1 Workspace window

The workspace window displays information about the organization of your workspace, projects and files. Context menus for the different items in a workspace provide commands to configure the way projects are built.

Output window

The output window displays messages from tools that you use to build your projects, as well as the results of textual searches performed using the Find in Files (see page 164) tool.

Messages in the output window that link to files appear as hyperlinks (blue and underlined by default). You can click on those messages to open relevant files at the specific locations.

MDI area

The MDI area hosts instances of the code editor, as well as other windows such as the CDFGViewer. You can manage the stack of windows in the MDI area with commands in the Window menu of the menu bar.

7.3 Creating a project

1. Select File > New > Project... from the menu bar.
2. Select an appropriate project type from those listed in the Project Types pane of the dialog.
3. Enter the name and choose a directory for your project.
4. Select whether you wish to add the project to the current workspace

   OR

   Create a new workspace.

   If you are creating a new workspace and project at the same time, the workspace will have the same name as the project and will be placed in the same directory by default. If you intend to have a multi-project workspace, you should select the option to create the workspace in a separate directory.

7.4 Adding files to a project

You can add existing source code files or create and add new files to a project in Agility.

- Use commands in the Project > Add to Active Project menu to add files to the ***Active Project*** (see page 163).
- Right-click the project icon in the ***workspace window*** (see page 155) to add files to any of the projects in the workspace.
- Right-click a folder icon to add files to a specific folder within a project.

Changing file types
Agility will use the file name extension to determine the type of a file added to a project. If you need to change the type of a file in a project, use the Properties command from the file icon's context menu within the workspace window.

7.5 Removing files from a project

You can remove a file from a project by:

- Selecting it in the workspace window and pressing Delete.

  OR
- By using the Remove command in the file icon's context menu.

Removing a file from a project will not delete the file from your computer's file-system.

7.6 Adding a project to a workspace

To add an existing project to the current workspace:

1. Select Project > Insert Project into Workspace...
2. Locate the project file (`.chp`) you wish to add the workspace and click OK.

7.7 Configuring a project

A project stores settings for the tools that are used to build it. Collections of settings are organized into configurations. You can switch between project configurations to change the way a project is built. Different configurations within the same project can direct the build process to execute the Agility tools with different arguments, or to execute external third party tools such as the GNU C++ compiler.

Agility default configurations
Agility projects will initially contain configurations with the following names:

- SystemC
- EDIF
- VHDL
- Verilog
- OSCI(win32-msvc.net)
- OSCI(linux-g++)

These default configurations direct the build in the following ways:

SystemC	Synthesize your SystemC into RTL (Register Transfer Level) SystemC. This can be used to verify the synthesis process using the same testbenches you use to verify your original SystemC.
EDIF	Synthesize your SystemC into an EDIF format netlist which is ready to be fitted / placed and routed for a specific device.
VHDL	Synthesize your SystemC into RTL VHDL, which can be simulated or synthesized further using third-party VHDL tools.
Verilog	Synthesize your SystemC into RTL Verilog, which can be simulated or synthesized further using third-party Verilog tools.
OSCI(win32-msvc.net)	Compile and link your SystemC code against the OSCI simulator on a Windows platform using the Microsoft Visual Studio .net C++ compiler.
OSCI(linux-g++)	Compile and link your SystemC code against the OSCI simulator on a Linux platform using the GNU C++ compiler.

7.7.1 Modifying configurations

You can modify the settings in the default configurations of your project to match your needs, or you can create new configurations by making copies of the existing configurations.

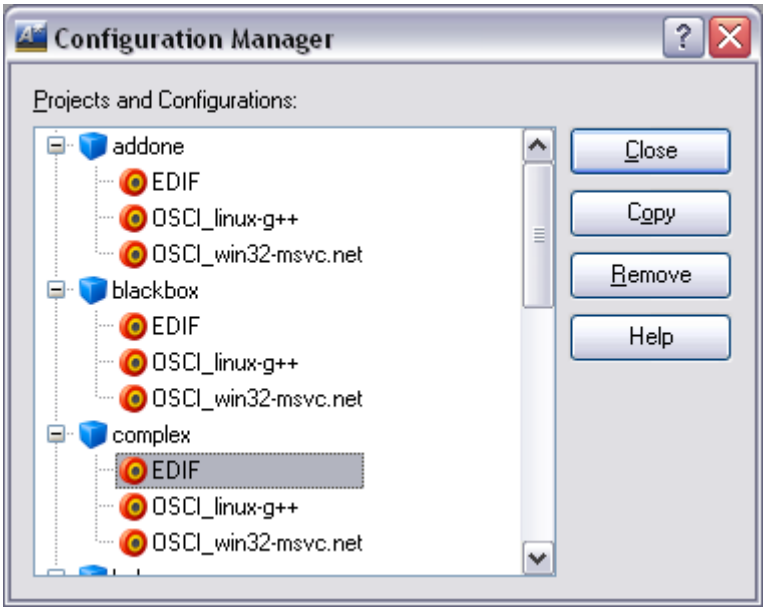
To create a new configuration for a project:

1. Select Project > Manage Configurations...

2. Select the configuration to copy with the mouse or the ↑/↓ keys.
3. Click Copy.
4. Enter the name for your new configuration.

To delete a configuration from a project:

1. Select Project > Manage Configurations...
2. Select the configuration to delete.
3. Click Delete.



You cannot copy configurations between different projects.

You cannot remove a lone configuration from a project.

7.8 Viewing and editing project settings

To view and edit a project's settings:

- Right-click the project icon, and select Settings... in the context menu
  OR
- Select Project > Settings... from the menu bar to access settings for the active project.

7.8.1 Changing the configuration displayed

The Settings dialog displays one configuration at a time. Change the configuration displayed by using the Configuration combo box at the top left of the Settings dialog.

Changing the item whose settings are displayed

If the Settings dialog is open you can change the item whose settings are displayed by using the mouse to select another icon in the workspace window.

Configuration settings for a specific file

You can change the build settings that apply to a specific file, although in most cases it is convenient to allow files in a project to share the same settings. Values shared between files are stored in the project settings.

Viewing and editing settings for a specific file
Right-click the file icon, and select Settings... in the context menu

Shared settings:

> Settings for a file that do not share the common value stored in the project are shown in bold font.
>
> You can restore the value of a File setting to the shared value by choosing the <Inherit from Project...> option in the combo box menu.

Editing configuration settings

The settings in a configuration are organized into different categories according to their function.

The category selected in the navigation window on the left controls which settings are visible in the settings list window on the right.

Settings may be disabled if they depend on another setting holding a certain value. Disabled settings are greyed out. The following settings are only enabled for certain output formats.

Setting	***Enabled for** Output format **settings***
VHDL/Verilog style	VHDL, Verilog, Verilog2001
Generate estimation information	EDIF
No Hierarchy	SystemC, VHDL, Verilog, Verilog2001
Enable Technology Mapping	Any value other than SystemC

Changing a configuration setting:

1. Select the category in the navigation window
2. Select the setting to edit by using the mouse or the Up/Down keys

3. A description of its function will be displayed below the settings list window and its value will be shown in an editable combo box.
4. Type in a new value for the setting,

   OR

   Select values from the combo box menu (click the combo box down arrow, or use the F4 key to show this menu).

If you enter an invalid value for a setting, the original value will be restored when you finish editing (by selecting another setting from the list or pressing Enter).

Some settings have extended editors to help you specify long or complicated values. You can access these editors by choosing the <Edit...> option in combo box menu.

7.9 Specifying the include path

Files that are included at the source level using the `#include` preprocessor directive are searched for in the following order:

1. Directory containing the source file using the `#include` directive (if the directive uses quotes rather than angle brackets).
2. Directories specified in the configuration settings for the file (in the order specified).
3. Directories specified in the options dialog available by selecting Tools > Options from the menu bar.

7.9.1 Using file and directory macros

The IDE supports a number of macros that you can use to compose values for settings that require file or directory names. Write the macros in the form ***$(Macro***) where ***Macro*** refers to one of the expressions listed below:

Macro	***Description***
`$(IntDir)`	Path to the directory specified for intermediate files, relative to the project directory.
`$(OutDir)`	Path to the directory specified for output files, relative to the project directory.
`$(ConfigurationName)`	The name of the project configuration.
`$(WorkspaceName)`	Workspace file name, excluding file extension.
`$(WorkspaceFileName)`	Workspace file name, including file extension.
`$(WorkspaceExt)`	Workspace file name extension.
`$(WorkspaceDir)`	Full path to directory in which workspace resides.
`$(WorkspacePath)`	Workspace file including full path and extension.
`$(ProjectName)`	Project file name, excluding file extension.

`$(ProjectFileName)`	Project file name, including file extension.
`$(ProjectExt)`	Project file name extension.
`$(ProjectDir)`	Full path to directory in which project resides.
`$(ProjectPath)`	Project file including full path and extension.
`$(InputName)`	Input file name, excluding file extension.
`$(InputFileName)`	Input file name, including file extension.
`$(InputExt)`	Input file name extension.
`$(InputDir)`	Full path to directory in which input file resides.
`$(InputPath)`	Input file including full path and extension.
`$(TargetName)`	Target file name, excluding file extension.
`$(TargetFileName)`	Target file name, including file extension.
`$(TargetExt)`	Target file name extension.
`$(TargetDir)`	Full path to directory in which the target file will reside.
`$(TargetPath)`	Target file including full path and extension.

`IntDir`, `OutDir`, and the `Target...` macros are generated from the values of settings in a configuration. You should exercise care not to define values in terms of themselves. The following table shows which settings these macros depend on:

Macro	***Depends on the configuration setting***
`$(IntDir)`	Intermediate directory
`$(OutDir)`	Output directory
`$(Target…)`	Output directory and Output file

For example, the value of setting Output file, (in the Linker > Output node), should not include any of the `Target...` macros.

7.10 Custom build steps

Custom build steps allow you to run external tools during a build in addition to the Agility compiler. You can specify custom build steps for:

- a project
- a file

Custom build steps can invoke multiple commands. Use the extended editor for the Batch commands setting and place each command on a separate line.

7.10.1 Project custom build steps

Custom build steps for projects are executed after the link stage.

If you specify the output file for the custom build step, the step will be run only when its output is out of date with respect to the target of the link stage. Otherwise, the step will be run each time you do a build.

A custom build step for a project is typically used to:

- invoke vendor's fitter / place and route tools on a netlist,
- invoke 3rd party synthesis tools on VHDL or Verilog output.

File custom build steps

A custom build step for a file is executed instead of a compile step (the compiler may be Agility or one specified as an external tool).

Files that are not passed directly to the regular compile tool (such as header files) can still have custom build steps.

A custom build step for a file is typically used to update another source file within the same project. This can be useful if you wish to use a script or macro processor to generate source code.

You must specify the output of a custom build step for a file or it will not be run.

7.10.2 Running user specified tools instead of Agility compiler

The IDE can run external tools (such as the GNU C++ compiler and linker) to build a project. This is useful for generating a machine executable model of your SystemC source code.

The default configurations, OSCI(win32-msvc.net) and OSCI(linux-g++), contain settings that instruct the build to use external tools to compile and link source files in a project. You can use these configurations as a base for creating configurations that run your own choice of tool.

Running user specified tools instead of Agility compiler

You must set up the following configuration settings to use external tools in a project

1. Set User specified tools to True in the top level General category for your configuration.

   Changing the value of this setting affects which other settings are available. The effect is not visible until you have committed the change by clicking Apply or OK.

2. Set the Input file type setting in the top level General category to an appropriate value.

   The IDE uses this setting to determine which files in the project it should generate explicit build rules for. It can then exclude files which are built implicitly, such as header files.

3. Specify command lines and outputs for your chosen compiler and linker.

The IDE evaluates the return code of an external process when performing a build, and stops the build if an external process reports failure.

Using file and directory macros to generalize settings

Use file and directory macros to generalize settings for command lines and output files for external tools. This allows command lines to be shared between many different source files in a project. For example:

```
gcc -c  -o  "$(IntDir)/$(InputName).o"     "$(InputPath)"
```

[command] [output flag and file name] [input file name]

The Command line setting for the link stage supports a special file macro `$?` that expands to the list of files generated by all the compile steps. You can use this to specify the inputs to a link step. Text placed immediately before the `$?` macro will be inserted before each of the file names to which the macro expands.

Example:
If "`$?`" expands to "`file1.obj file2.obj file3.obj`",

"`link /Tp$?`" will be expanded to "`link /Tpfile1.obj /Tpfile2.obj /Tpfile3.obj`"

7.11 Building a project

The active project is built using the active configuration when:

- You click on the Build button in the Build toolbar.
- You select Build > Build from the menu bar or press the F7 key.

You can build any of the projects in a workspace using the Build command in the context menu for a project's icon in the workspace window.

When you build a project, the IDE will:

1. Calculate the dependencies between files in the project.
2. Compare the date stamps for source, object and target files.
3. Execute the appropriate tools required to make the output files up-to-date with respect to the input files.

You can force the IDE to build files that are already up-to-date by selecting Build > Rebuild All from the menu bar.

You can make the IDE delete intermediate files from the build (such as object files) by selecting Build > Clean from the menu bar. After doing this, all the files in a project will be rebuilt on the next build. Executing Clean and then Build is equivalent to executing Rebuild All.

7.11.1 Setting the active project and the active configuration

You can set the active project and the active configuration using the Project > Set Active Project, and Project > Set Active Configuration menus. Alternatively you can use the combo boxes in the build toolbar.

7.12 Using the report view

You can open, browse and navigate XML report files generated by the Agility compiler in the IDE. Select File>Open from the menu bar and navigate to the report file you wish to view. By default, report files are generated in the output directory for a build.

During a build, the Agility compiler will produce the message `Report file generated`. You can click on this message to automatically open the report in the IDE.

The report view displays messages generated by the Agility compiler. Where those messages correspond to positions in your source code, the report view will display hyperlinks that you can use to jump to the corresponding positions in specific source code files.

7.13 The code editor

The code editor provides several facilities to assist you in reading and writing program source code, including:

- Syntax colouring for keywords, comments and other text categories.
- Automatic line indentation.
- Automatic brace matching.
- Code folding.
- Indentation guides.
- Bookmarks.
- Search and replace.

To change the default settings for these features, select Tools > Options from the menu bar. Select General>Fonts and Colours node or the Code Editor node and make the desired changes.

7.14 Find in Files

You can use the Find in Files tool to perform a text search in multiple files. Select Edit > Find in Files... from the menu bar to raise the Find in Files dialog.

You can use regular expressions to specify a textual pattern to match. All matches are reported in one of the Find in Files tabs of the output window.

Double click on the match message to go the match in the code editor (this opens the file containing the match if necessary).

7.15 Simulating with the OSCI SystemC libraries

If you have built your project against the OSCI ***SystemC libraries*** (see page 162) you can run the generated executable file within the IDE.

The default configurations, OSCI_win32-msvc.net and OSCI_linux-g++, contain settings that instruct the IDE to launch their respective build targets as an external process when you select Debug>Run from the menu bar.

The following configuration settings alter the way external applications are run:

- Command line in the Execution>General category specifies the name of the application to be run and any arguments it requires.
- Working directory specifies the directory in which the application will be run.

Windows simulation
Under Windows, you may find it convenient to run OSCI simulations in a command shell, as this will enable you to input data required by the simulation.

To run the application in a command shell, precede the application file name with `cmd /k start /wait` in the Command line setting.

Linux simulations
Under Linux, you may find it convenient to run OSCI simulations in an X-Terminal (`xterm`), as this will enable you to input data required by the simulation.

To run the application in an `xterm`, precede the application file name with `xterm -e` in the Command line setting.

Ending simulations
To stop an external process before it runs to completion, you should type Ctl-C into the `xterm` or command prompt. You may also select Debug>Stop from the menu bar, but this may prevent the process from terminating gracefully.

7.16 The simulation path

If you are creating a testbench you can refer to the OSCI simulator directory in your Makefile or Visual Studio projects using the CELOXICA_OSCI_SIM environment variable. This is set up when Agility is installed in Windows. In Linux the System-C Debian or RPM package can be installed in standard system paths, so this environment variable is not needed.

8 Celoxica CDFGViewer

Agility includes a viewer to display the Control and Data Flow Graph (CDFG) of your design.

You can display sections of the graph and relate them back to the source code.

8.1 Agility CDFGViewer overview

The CDFGViewer allows you to view and navigate the CDFG generated by the Agility Compiler.

This is useful because:

- Viewing your design in terms of data-flow and control-flow can help your understanding of your design
- The graph indicates how Agility has synthesized your code: where registers have been placed and operations shared

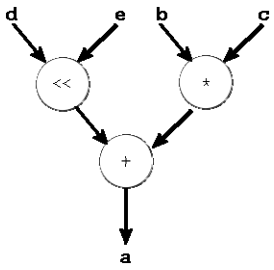
8.1.1 What is a CDFG?

A CDFG is a graph representation of your design. It shows the data-flow and control-flow in your program.

Data-flow
The statement

```
a = (d << e) + (b * c);
```

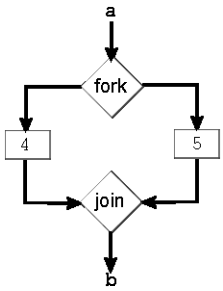
is an example of data-flow. It can be represented by the following graph.



Control-flow
SystemC control-flow statements (such as `if` and `switch)` can be represented in a CDFG. For instance the code snippet

```
if( a ) b = 4;
   else b = 5;
```

can be represented by the following graph:



8.1.2 The Agility CDFG

The Agility CDFG is composed of:

- Hierarchical groups containing further groups and nodes
- Nodes
- Directed edges

The CDFG is drawn on a *canvas*.

8.1.3 Groups

A group has a type, which corresponds to the use of a language construct within your SystemC design. For instance, an `if`-statement generates an `if` group. Each group type has a one-to-one correspondence with a language construct, except for loop-type groups. `while`-loops, `do-while`-loops and `for`-loops all generate loop groups (if they have not been unrolled). In each case, the loop exit condition is turned into an explicit `if` group.

8.1.4 Nodes

Nodes correspond to:

- operations
- control-flow
- constants
- reads from and writes to primitives (SystemC or Agility-specific). Note that the primitives themselves are not displayed, only interactions with them

For instance, when Agility synthesizes the `if` statement

```
if (b==0)
    a = b+1;
else
    a = b-1;
```

the associated CDFG will have

- three operation nodes, (==, +, −)
- two control nodes

a fork node to select between branches

a node to resolve the different assignments onto the variable `a`.

- three constant nodes.

8.1.5 Edges

Edges allow data (or control) to flow from one node to another.

There are two types of edge:

- Data-flow edges carry data between data-flow nodes.
- Control-flow edges carry control-flow

There are two further types of control-flow edge:

- Clock edges connect all nodes in the same clock domain.
- Reset edges connect all nodes with the same reset sensitivity.

You can hide clock and reset edges in the CDFG. (see ***Viewing clock edges*** (see page 176) and ***Viewing reset edges*** (see page 176))

8.2 Elements of the Agility CDFGViewer

The groups, nodes and edges are displayed in different colours according to their purpose.

8.2.1 Group rendering

Groups are rendered as boxes with a coloured outline. Collapsed groups are shown as grey boxes that hide their contents. The colours used for the outline correspond to different group types.

Group Type	*Origin in source code*	*Colour*
Exit	`break` or `continue` statements	Black
Function	function call	Light-blue
If	`if` statement	Dark-blue
Loop	any loop (`while`, `do-while` or `for`) statement	Lime-green
Module	`sc_module` instantiation	Dark-blue
Seq	any series of statements	Light-grey
Switch	`switch` statement	Light-grey
Wait	`wait` statement	Red

8.2.2 Node rendering

Nodes are rendered with different shapes and different colours depending on their type. There are many ***node types*** (see page 177), but most are self-explanatory. The most important and common types are described in the sections:

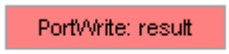
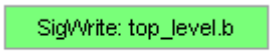
- Data flow nodes
- ***Control flow nodes*** (see page 170)
- ***Other nodes*** (see page 170)

Nodes showing shared operations are highlighted with a dark-yellow outline. For example:

The displayed CDFG is augmented with information showing which operations are shared.

Data-flow nodes

Data-flow nodes show the path of data between operators and port reads and writes.

Node Type	*Origin in source code*	*Example of shape and colour*
Const	Use of a constant	
Binary operators	Use of a binary operator (e.g. `+`, `*`, `<<`, `!=`, `<` …)	
Unary operators	Use of a unary operator (e.g. `!`, `~`, `-`…) and casting	
DataWait	Use of `wait` statement	
Port Read/Write	Use of ***port***`.write` or ***port***`.read`	
Signal Read/Write	Use of ***signal***`.write` or ***signal***`.read`	

Control-flow nodes

Control-flow nodes show the control-flow routes. For example:

Node Type	*Origin in source code*	*Example of shape and colour*
Start	Starts a state-machine in a thread	
ClfDecode	Associated with any conditional	
Ctrue	Associated with control flow	
Switch	Associated with switch statements	

Other nodes

The Agility Compiler adds nodes internally. They do not correspond to statements in the design, but make explicit the control-flow between statements.

Node Type	*Explanation*	*Example*
Group Control	Used as common entry point to all groups for variables and control path	
State Resolve	Used to resolve branches in if-statements and provides implicit loop back edges	

8.2.3 Edge rendering

Edges are rendered with different colours depending on their type.

You can group data edges together to reduce the number of edges displayed (see ***Grouping and ungrouping data edges*** (see page 175)).

Edge Type	*Origin in source code*	*Colour*
Data (temporary)	Created between operations in an expression	Light-red
Data (variable)	Corresponds to variable usage	Red
Control	Connects control statements	Blue
Reset	Connects operations and registers with the same reset sensitivity to the reset control	Light-grey
Clock	Connects operations and registers in the same clock domain to the clock port	Grey

8.3 Using the CDFGViewer

The CDFGViewer allows you to explore the CDFG by expanding and collapsing groups to display or hide the groups' contents. By default, groups are collapsed. Further information is displayed through tool tips.

8.3.1 CDFGViewer display options

CDFGViewer display options define how information is presented in the CDFGViewer. From the GUI, select Tools>Options. In the dialog, select the CDFGViewer node under the plugins folder. The different option settings are available via tabs.

- Groups tab
- Edges tab
- Layout tab

Groups tab

The Groups tab specifies how groups are expanded.

Item	*Description*	*Default*
Always expand 'seq' groups	When displaying graphs expand every seq group encountered, unless the user has explicitly closed the group.	True
Always expand 'wait' group	When displaying graphs expand every wait group encountered, unless the user has explicitly closed the group.	False

Edges tab

The Edges tab specifies how edges are displayed.

Item	*Description*	*Default*
Don't display clock edges	When displaying graphs, do not display clock edges between nodes. Note: There are many clock edges in a typical design. Displaying clock edges may create cluttered graphs that are slow to draw	True
Don't display reset edges	When displaying graphs, do not display reset edges between nodes. Note: There are many reset edges in a typical design. Displaying reset edges may create cluttered graphs that are slow to draw	True
Don't display multiple data edge	When displaying graphs, merge data edges between nodes. All control edges are still displayed.	True

Layout tab

The Layout tab specifies how the canvas is animated.

Item	*Description*	*Default*
Animate Layout	When displaying graphs, animate the collapsing and expanding of groups.	True

Animation of large graphs can be slow. Experiment with turning the option off.

8.3.2 Creating a file for the CDFGViewer

The CDFGViewer displays `*.cgv` files

- You can produce a CDFGViewer file from the Agility command line or the IDE.

From the IDE

- Open an Agility project.
- Select Project>Settings
- Under the SystemC configuration properties select the Linker node.
- Type the name of the `cgv` file in the Output CDFG data field.
- Press OK.

You will then need to (re-)build the Agility project.

From the command-line

Use the `--output-cdfg` option. E.g.

```
$ agility --output-cdfg my_cdfg.cgv test.sc.cpp
```

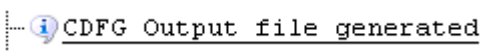
will compile `test.sc.cpp` and produce the file `my_cdfg.cgv`.

8.3.3 Viewing a CDFGViewer file

Created from the IDE

If you used the Agility IDE to synthesize your design, a message will be produced in the output pane.

You can click on the message to load the CDFGViewer file into the IDE.



Created from the command-line

If you synthesized your design by invoking Agility from the command-line, you can still open the `cgv` file from the IDE.

- Select File>Open
- Select the correct `cgv` file.

When you open a `cgv` file, the top-level group is shown collapsed.

Navigating the CDFG

- Expanding a group
- Collapsing a group
- Scrolling over the graph
- Invoking the birdseye
- Displaying the CDFG for a section of code
- Displaying information about an edge
- Displaying sharing information
- Finding source code that generated a node
- Disabling animation
- Ungrouping data edges
- ***Viewing clock edges*** (see page 175)
- ***Viewing reset edges*** (see page 176)

Expanding a group

You can expand any collapsed group by

- clicking on the + sign in the top-left corner of the box that represents the group.
- double-clicking anywhere on the box.

Collapsing a group

You can collapse any expanded group clicking on the — sign in the top-left corner of the box.

Scrolling over the entire graph

As designs are usually large, it is generally impossible to display the entire graph on screen. In order to scroll over regions of the graph that are not displayed you can:

- use the scroll bars at the right and bottom of the canvas.
- drag the graph by clicking on an empty region of the canvas and dragging the mouse.
- hold down the mouse whilst tracking over the birdseye canvas to scroll around the CDFG.

Invoking the birdseye

The birdseye view of the CDFG is a small pane that can be used to display the entire graph.

1. Open a `cgv` file.
2. Right-click on any of the windows.
3. Select Birdseye.

The birdseye can be closed by clicking on the X in the top-right corner.

It will automatically be closed when `cgv` file is closed.

Displaying the CDFG for a section of source code

1. Open the source file and associated project you are interested in. Make sure that the project has been built and is up-to-date.
2. Open the associated `cgv` file.

3. In the source code editor, highlight the section of source code you are interested in. (e.g., a single function).
4. Right-click to invoke the context-menu.
5. If the region of source code you have selected corresponds to a valid subset of the entire CDFG, Display Graph will be enabled in the menu. If this option is greyed-out, select another region of source code.

Finding the source code that generated a node

Double-click the node to open the associated source file in the editor. The line of source code corresponding to the node is highlighted.

Displaying information about an edge

To keep the graph clear, information about an edge is displayed in a tool tip. The tool tip is shown when the mouse hovers within a short distance of the edge.

Edge type	***Information***
Expression	Signed/unsigned and bitwidth
Data	Variable name
Control	(none)
Clock	(none)
Reset	(none)

Displaying shared operations

Some operations may have been shared during synthesis.

Nodes representing shared operations have a dark-yellow outline. For example:

Showing all operations sharing resource

Click a highlighted node to link all operations implemented on the same resource with a green arc.

Click again on any of the shared nodes to remove the arc.

Disabling animation

If the drawing of the graph is slow you can disable the animation feature to improve performance.

1. Select the Tools>Options dialog box.
2. Select the CDFGViewer node under the plugins folder.
3. Select the Layout tab.
4. Clear the Animate Layout checkbox.

Grouping and ungrouping data edges

You can group data edges together to reduce the number of data edges between nodes. Data edges can only be grouped if their source node and sink node are identical.

1. Select the Tools>Options dialog box.
2. Select the CDFGViewer page under the plugins folder.
3. Select the Edges tab.
4. Clear the Don't display multiple data edges checkbox.

Viewing clock edges

You can view the edges connecting a clock port with all nodes that it is driving.

1. Select the Tools>Options dialog box.
2. Select the CDFGViewer page under the plugins folder.
3. Select the Edges tab.
4. Clear the Don't display clock. edges checkbox.

Viewing reset edges

You can view the edges connecting a reset port with all nodes that it is driving.

1. Select the Tools>Options dialog box.
2. Select the CDFGViewer page under the plugins folder.
3. Select the Edges tab.
4. Disable the Don't display reset edges checkbox.

8.3.4 CDFGViewer Module Call Graph

The CDFGViewer Module Call Graph allows users to see the modules instantiated and functions called in their code. The function call hierarchy is represented as a tree, with the top-level module as the root.

Elements of the Module Call Graph

The Module Call Graph shows three elements in the CDFG:

- Instantiated modules
- Instantiated processes
- Function calls

Type	***Origin in Code***	***Icon***
SystemC module	Module instantiation	
`SC_METHOD`	Function declared as an `sc_method`	
`SC_THREAD`	Function declared as an `sc_thread`	
Function	Function call	

Using the Module Call Graph

The Module Call Graph opens when a CDFG is loaded into the Agility IDE. It is automatically closed when the CDFG is closed.

It appears in the left hand pane. You can switch between the Module Call Graph view and the files and directories tree view by clicking the Module Call Graph tab or the Files tab.

Displaying sections of the CDFG

You can display the section of the CDFG that corresponds to a function call by:

- Double-clicking on the item in the Module Call Graph.
- Right-clicking-click on the item and selecting Draw Selected Group from the context menu.

Finding the source code that corresponds to a function call

Right-click an item in the Module Call Graph and select Go To Usage from the context-menu.

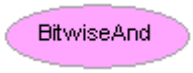
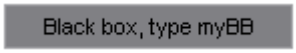
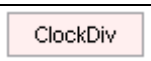
The source code editor opens with the line of source code corresponding to the function call highlighted.

8.4 Appendix: complete list of nodes

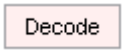
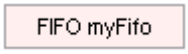
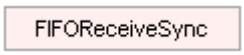
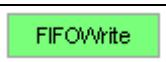
The nodes are colour-coded.

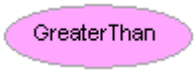
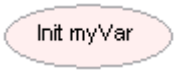
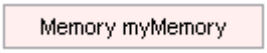
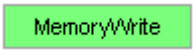
- Control flow: blue
- Reads and writes: green
- Data flow nodes have various colours depending on function.

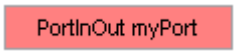
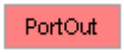
This appendix provides a complete list of all nodes. Most of them are self-explanatory. Some are generated by the compiler and are included in here for the sake of completeness (they have no direct relationship to source code). If the relationship to source code is not obvious, it is described in the list below.

Node type	*Example*
ArrayRead	
ArraySplice	
ArraySplit	
ArrayWrite	
BitwiseAnd	
BitwiseOr	
BitwiseXor	
Black box (Instantiation of `ag_foreign_module`)	
CAnd	
Cast	
CFalse	
CFF	
ChannelRead	
ChannelWrite	
ClfDecode (Generated by every `if` statement)	
CJoin (Generated by `if` and loop statements)	
ClockBuf	
ClockDiv	
CNot	

Concat	
Const	
COr	
CTrue	

DataMux	
DataWait (Generated for each variable registered at a `wait` statement)	
Decode	
DHold	
Div	
End	
Equals	
FIFO	
FIFORead	
FIFOReadAck	
FIFOReadData	
FIFOReadReq	
FIFOReceiveSync	
FIFOSendSync	
FIFOWrite	
FIFOWriteAck	
FIFOWriteReq	

GreaterThan	
GroupControl (Aids understanding of control flow in CDFG)	
GTE	
Init	
Invert	
Left Shift	
LessThan	
LTE	
Memory	
MemoryRead	
MemoryWrite	
Minus	
Mod	
NotEquals	

Plus	
PortIn	
PortInOut	
PortOut	
PortRead	
PortWrite	
Range	
Register	
ResetBuf	
SignalRead	
SignalWrite	
SigState	
SplitWire	
Start	
StateResolve (Aids understanding of control flow)	
Switch	
Times	
UnaryMinus	
Wait (Generated at each `wait` statement)	

9 Agility command-line compiler reference

This is a reference to the command-line compiler options.

9.1 Using the command-line compiler

The Agility Compiler can be invoked from the command-line as well as from the Agility IDE. This allows you to:

- use alternative tools to edit source code
- use esoteric Agility features
- automate build processes by means of batch files, scripts or makefiles.

The command-line options are the equivalent of the settings found in the Project settings dialog.

The Agility executable is named `agility` in Linux and `agility.exe` in Windows. The options are the same for both. In the following examples the dollar symbol (`$`) is used to represent the prompt. Agility is invoked in the following way:

`$ agility` ***options input-files***

9.1.1 Command-line options

Agility options begin with a double hyphen.

Some common options also have a short version consisting of a hyphen and a single letter.

Many options take one or more arguments. It is optional to have an equals sign between the option and its argument.

For a summary of the options execute:

```
$ agility --help
```

General options

`--help` `-h`	Display a help message summarizing the options to the compiler
`--version` `-v`	Show version information about the Agility compiler. This consists of a major and minor release number as well as a build number. This information should always be supplied when contacting Celoxica support.

Pre-processor options

Agility's built-in preprocessor accepts the following options. To view the pre-processed output, use the `-E` ***output option*** (see page 184).

`--include-path` ***directory*** `-I` ***directory***	Add a directory to be searched for header files.
`--no-default-include`	Disable the built-in search paths for header files. This prevents Agility finding essential system header files unless alternative paths are specified explicitly.
`--preprocessor-define` ***macro*** `-D` ***macro***	Define a macro to be used by the pre-processor. The value is optional, so `-DSOMEMACRO` is valid as well as `-DSOMEMACRO=1`.
`--preprocessor-undefine` ***macro*** `-U` ***macro***	Un-define a pre-processor macro, for example `-USOMEMACRO`.

Output options

Output options instruct Agility's synthesis engine to produce a particular type of output. If you do not specify a name for the output file, Agility generates a default name based on the input file.

`--edif`	Output an ***EDIF netlist*** (see page 130). A family option will be required. The default suffix is `.edf`.
`--vhdl` `--systemc`	Output ***RTL SystemC*** (see page 139) netlist, the default extension is `.sc.cpp`.
`--verilog`	Output ***Verilog*** (see page 135). The default file suffix is `.v`.
`--verilog-2001`	Output Verilog IEEE 1364-2001. The default extension is `.v`.
`--vhdl`	Output ***VHDL*** (see page 132). The default extension for this type of file is `.vhd`.
`--compile-only` `-c`	Compile only, output an object file. The usual file extension for Agility object files is `.ago`.
`--preprocess` `-E`	Pre-process only. Outputs a pre-processed version of the original SystemC design. Useful for debugging problems with macro expansion.
`--output-file` ***file*** `-o` ***file***	Specify the name for the output file from Agility.

Compiler options

`--warning-level` ***level*** `-W` ***level***	Specify the warning level. Valid warning levels are 0-4 (default 2, 0 no warnings, 4 all warnings).

Input options

Input options force a file to be interpreted as a particular type, irrespective of its file extension. Otherwise, Agility assumes the file type from the extension. It is an error to give a file with an unknown or missing extension and no input option.

`--agility-library-file` ***file***	Treat the given file as an Agility library file even if it is not suffixed with `agl`.
`--agility-object-file` ***file***	Treat the given file as an Agility object file even if it is not suffixed with `ago`.
`--systemc-source-file` ***file***	Treat the given file as SystemC source even if it is not suffixed with `cpp`.

Reporting options

`--estimate-logic-size`	Output logic estimation information. The information includes annotated source code and is presented as a number of HTML files that can be viewed in the Agility IDE.
`--output-cdfg` ***file***	Output a CDFG that can be viewed using the ***CDFG viewer*** (see page 166) in the Celoxica Development Environment. The usual file extension for this type of file is `cgv`.
`--report` ***file***	Output a report file in an XML format. The compilation information is provided in a structured form suitable for display in Agility IDE.

Synthesis options

`--alu-limit` ***alumap=number***	Limit instances of a specified ALU to a given number.
`--alu-mapping` ***bool***	Map to ***embedded ALUs*** (see page 128) (default 1 – enabled).
`--fast-carry-chains` ***bool***	Use fast carry chains, (default 1 – enabled).
`--hdl-style` ***style***	Set the VHDL or Verilog style to ***one tailored for a particular tool*** (see page 140). The style can be one of: `ActiveHDL`: Aldec Active-HDL `DesignCompiler`: Synopsys Design Compiler `ModelSim`: Model Technology ModelSim `Precision`: Mentor Graphics Precision `Synplify`: Synplicity Synplify `Generic`: Other tools (default)
`--netlist-opt` ***arg***	Optimization mode. Argument can be `speed` or `area` (default is `area`).
`--no-hierarchy`	Output flattened VHDL, Verilog or SystemC.
`--pipeline-ram-access` ***bool***	Pipeline synchronous RAM accesses (default 1 – enabled).
`--ports-as-pins` ***bool***	Build top-level ports as pins (default 0 - disabled).
`--retiming` ***bool***	Use retiming (default 0 – disabled).
`--technology-mapping` ***bool***	Use technology mapping (default 0 – disabled).
`--timing-constraints` ***bool***	Use timing constraints (default 0 – disabled).
`--family` ***arg*** `-f` ***arg***	Specify the target family, e.g. `XilinxVirtex4`
`--part` ***arg*** `-p` ***arg***	Specify target chip number., e.g. `V1000BG560-4`

Supported families

Family name	*Description*
`Actel500K`	Actel ProASIC series FPGAs
`ActelPA`	Actel ProASIC+ series FPGAs
`AlteraFlex10K`	Flex10K series Altera PLDs
`AlteraFlex10KA`	Flex10KA series Altera PLDs
`AlteraFlex10KB`	Flex10KB series Altera PLDs
`AlteraFlex10KE`	Flex10KE series Altera PLDs
`AlteraApex20K`	Apex 20K series Altera PLDs
`AlteraApex20KE`	Apex 20KE series Altera PLDs
`AlteraApex20KC`	Apex 20KC series Altera PLDs
`AlteraApexII`	Apex II series PLDs
`AlteraMercury`	Altera Mercury series PLDs
`AlteraStratix`	Altera Stratix PLDs
`AlteraStratixII`	Altera Stratix II PLDs
`AlteraStratixGX`	Altera Stratix GX PLDs
`AlteraCyclone`	Altera Cyclone PLDs
`AlteraCycloneII`	Altera Cyclone II PLDs
`AlteraExcaliburARM`	Altera Excalibur ARM series PLDs
`XilinxVirtex`	Virtex Xilinx FPGAs
`XilinxVirtexE`	VirtexE Xilinx FPGAs
`XilinxVirtexII`	Virtex-II Xilinx FPGAs
`XilinxVirtexIIPro`	Virtex-II Pro Xilinx FPGAs
`XilinxVirtexIIProX`	Virtex-II Pro X Xilinx FPGAs
`XilinxVirtex4`	Virtex-4 Xilinx FPGAs
`XilinxSpartan`	Spartan Xilinx FPGAs
`XilinxSpartanXL`	Spartan-XL Xilinx FPGAs
`XilinxSpartanII`	Spartan-II Xilinx FPGAs
`XilinxSpartanIIE`	Spartan-IIE Xilinx FPGAs
`XilinxSpartan3`	Spartan-3 Xilinx FPGAs
`XilinxSpartan3E`	Spartan-3E Xilinx FPGAs
`XilinxSpartan3L`	Spartan-3L Xilinx FPGAs

Optimization options

`--repeated-conditional-rewriting` ***bool***	Control repeated-conditional-rewriting optimizations (default: 0 – disabled). If enabled this option can substantially increase compile time.
`--tree-balancing` ***bool***	Control tree-balancing optimizations (default: 1 – enabled).

(Further information about these options is available in the description of the Linker node: Project settings in the Agility IDE.)

9.1.2 Example: Command line

```
$ agility --compile-only -o prog.ago prog.sc.cpp
```

```
$ agility --verilog-2001 --agility-object-file prog.ago
```

Compiles `prog.sc.cpp` to an Agility object file, then generate IEEE 1364-2001 Verilog from the resulting object file.

```
$ agility --no-hierarchy --vhdl prog.sc.cpp
```

Compiles `prog.sc.cpp` to flattened VHDL with the default output style and optimizations.

```
$ agility --systemc -o synth.sc.cpp prog.sc.cpp
```

Compiles `prog.sc.cpp` to ***RTL-level SystemC*** (see page 139), saving the output in a file named `synth.sc.cpp`.

```
$ agility --edif --family=XilinxVirtexII prog.sc.cpp
```

Compiles `prog.sc.cpp` to EDIF for the Xilinx VirtexII family of FPGAs.

9.2 Option dependencies

Some of the options only relate to a particular output format.

The different output formats are mutually exclusive.

Output format	*Relevant options*	

EDIF	`--estimate-logic-size`
	`--netlist-opt`
	`--alu-mapping`
	`--alu-limit`
	`--technology-mapping`
		`--timing-constraints`
		`--fast-carry-chains`
		`--pipeline-ram-access`
		`--ports-as-pins`
		`--repeated-conditional-rewriting`
		`--tree-balancing`

VHDL	`--no-hierarchy`
	`--hdl-style`
		`--ports-as-pins`
		`--repeated-conditional-rewriting`
		`--tree-balancing`

Verilog/Verilog 2001	`--no-hierarchy`
	`--hdl-style`
		`--ports-as-pins`
		`--repeated-conditional-rewriting`
		`--tree-balancing`

RTL SystemC	`--no-hierarchy`	`--ports-as-pins`
		`--repeated-conditional-rewriting`
		`--tree-balancing`

10 Agility examples

Agility comes with several examples.

- They can be compiled to run using the OSCI simulator using Visual Studio .NET or `gcc`.
- They can be synthesized to EDIF, Verilog, VHDL or RTL SystemC from Agility

They are supplied in a single Agility workspace (`examples.chw`). They are also accessible from a Visual Studio .NET 2003 solution file (`examples.sln`)

Each example comprises:

- Synthesizable SystemC source code and header files
- a simulation testbench in SystemC
- a synthesis top-level file

- an Agility project file
- a Visual Studio .NET 2003 project file
- a `makefile` for `gcc`

Before you can run the examples under Linux, you must run the shellscript `sh /usr/share/agility/copy_examples_to_home_directory.sh` to copy them to your home directory.

To simulate the examples within the Agility IDE, you need to ensure that you have the `CELOXICA_OSCI_SIM_ROOT` environment variable set up to point to the root directory of your SystemC 2.1 installation.

11 Agility examples supplied

The following SystemC examples are supplied with Agility:

`addone`

`sc_methods`

`reset`

`led`

`sevenseg`

`multiclock`

`blackbox`

`complex`

12 Agility Example: addone

Demonstrates:
Single top-level module, using a testbench in simulation

Description
This example shows a single `sc_module` with a single `sc_thread`. The thread reads continuously from an `sc_in` port. The value that is read from the port is incremented by one and written synchronously onto an `sc_out` port. The example is driven by a testbench in simulation, and the user can supply test values and watch the returned result on a console.

12.1 Building the addone example for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the addone project and select Set as StartUp Project.
3. Right-click on the addone project and select Project Only -> Build Only addone

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `addone` directory you created:
   `cd ~/agility_examples/addone`
3. Build the example:
   `make`
4. Run the example:
   `./addone`

12.2 Simulating the addone example

Once the simulation has started, the user is presented with a console and invited to supply a decimal value to the `addone` module. The incremented value is returned and displayed on the console. To stop the simulation the user can enter `-1`.

An output waveform file, `addone.vcd`, is created.

12.3 Building the addone example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the project and select Set as Active Project.
3. From the Build menu, select Build.

13 Agility Example: sc_methods

Demonstrates:

`sc_method`

Description:
This example is similar to the `addone` example. However, it contains no `wait()` statements as it uses an `SC_METHOD` instead of an `SC_THREAD`. Therefore, the hardware which is created is entirely combinational.

13.1 Building sc_methods for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the sc_methods project and select Set as StartUp Project.
3. Right-click on the sc_methods project and select Project Only -> Build Only sc_methods

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `sc_methods` directory you created:
   ```
   cd ~/agility_examples/sc_methods
   ```
3. Build the example:
   ```
   make
   ```
4. Run the example:
   ```
   ./sc_methods
   ```

13.2 Simulating the sc_methods example

`sc_methods` asks the user to input a number. It then adds one to that number and displays the result. Whilst this is a very simple demonstration, it shows how to create some combinational logic.

An output waveform file, `sc_methods.vcd`, is created, which shows the values on the ports from the simulation run.

13.3 Building the sc_methods example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the sc_methods project and select Set as Active Project.
3. From the Build menu, select Build.

14 Agility Example: led

Demonstrates:

- basic design layout
- declaring ports
- writing a design for simulation and hardware

Description:

The `led` example shows how to layout a SystemC design for Agility Compiler. Code is initially run from the `ag_main()` function where you must instantiate your top-level module.

`led.sc.h` declares the top-level `sc_module` and the ports, variables and functions within it. The constructor for this module runs the `sc_thread` and defines its sensitivity list.

`led.sc.cpp` defines the functions in the module. Note that the value of ***clock_rate*** varies depending on whether you are building for hardware or simulation. Because hardware runs at several megahertz, this causes the LEDs on the RC10 board to alternate every second. This would take a long time to simulate, however, so the LEDs change every 10 clock cycles. This is visible in the output waveform file.

The simulation testbench, `testbench.sc.cpp`, connects `sc_signals` to the ports in the ***led*** module. These signals are connected to a trace file so that their value can be tracked over time.

14.1 Building the led example for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the led project and select Set as StartUp Project.
3. Right-click on the led project and select Project Only -> Build Only led

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `led` directory you created:
   ```
   cd ~/agility_examples/led
   ```
3. Build the example:
   ```
   make
   ```
4. Run the example:
   ```
   ./led
   ```

14.2 Simulating the led example

`led` is a non-interactive example. It runs for 5ms and creates an output waveform file, `led.vcd`, which shows the values of the ports throughout the simulation run.

14.3 Building and running the led example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the led project and select Set as Active Project.
3. From the Build menu, select Build.
4. If you have an RC10 Celoxica board, you can use the file transfer utility (`FTU3`) supplied with Celoxica boards to download the example to it.

15 Agility Example: sevenseg

Demonstrates:

- writing for hardware
- arrays
- constraints file

Description

This application builds on the `led` example to show how to target a hardware platform by using a constraints file. It displays hex digits counting on an RC10 board.

`sevenseg.ucf` demonstrates how to link port names to physical pins on a board and how to set a target clock rate for the place and route tools.

The `digit_table` array is used to set the pin configuration for each seven-segment display from an index, 0 to 15.

15.1 Building the sevenseg example for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the sevenseg project and select Set as StartUp Project.
3. Right-click on the sevenseg project and select Project Only -> Build Only sevenseg

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `sevenseg` directory you created:
   ```
   cd ~/agility_examples/sevenseg
   ```
3. Build the example:
   ```
   make
   ```
4. Run the example:
   ```
   ./sevenseg
   ```

15.2 Simulating the sevenseg example

`sevenseg` is a non-interactive example. An output waveform file, `sevenseg.vcd`, is created, which runs for 10ms and shows the value of the ports over this time.

15.3 Building and running the sevenseg example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the sevenseg project and select Set as Active Project.
3. From the Build menu, select Build.
4. If you have an RC10 board, you can use FTU3 to download the example to it.

16 Agility Example: complex

Demonstrates:

- template classes
- operator overloading
- user-defined datatypes

Description

This example shows how the user can define template classes, allowing generic SystemC code to be written. In addition, operator overloading can be used to provide a consistent syntax for user-defined datatypes.

A testbench is supplied to drive the arithmetic behaviour of the datatype, and the user can supply test-vectors to verify that the arithmetic works correctly.

16.1 Building the complex example for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the complex project and select Set as StartUp Project.
3. Right-click on the complex project and select Project Only -> Build Only complex

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `complex` directory you created:

   ```
   cd ~/agility_examples/complex
   ```

3. Build the example:

   ```
   make
   ```

4. Run the example:

```
./complex
```

16.2 Simulating the complex example

When the simulation is started, the user is prompted to enter a complex number. The user should separate the two requisite decimal numbers with a space, before entering a second complex number in the same manner. Finally an operation to perform on the complex numbers should be chosen. The result of the operation on the two complex numbers will be displayed on the console.

16.3 Building the complex example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the complex project and select Set as Active Project.
3. From the Build menu, select Build.

17 Examples of Agility features

These examples show implementation of Agility extensions to the synthesizable subset:

- Maths library example
- Signal with default example
- Agility Example: multiclock
- Agility Example: blackbox
- RAMs example
- Agility Example: reset

17.1 Agility multiclock example

Demonstrates:
Using multiple independent clocks

Description:
This example builds on the `sevenseg` example to display two counters which run at independent speeds.

The two input clocks, `clock_in_0` and `clock_in_1`, are connected to different pins through `multiclock.ucf`.

17.1.1 Building the multiclock example for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the multiclock project and select Set as StartUp Project.
3. Right-click on the multiclock project and select Project Only -> Build Only multiclock

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `multiclock` directory you created:
   `cd ~/agility_examples/multiclock`
3. Build the example:
   `make`
4. Run the example:
   `./multiclock`

17.1.2 Simulating the multiclock example

`multiclock` is a non-interactive example. It creates an output waveform file, `multiclock.vcd`, which runs for 10ms and shows the value of all the ports over this time.

The simulation testbench creates two clocks for stimulating the threads, which run at intervals of 40ns and 100ns.

17.1.3 Building and running the multiclock example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the multiclock project and select Set as Active Project.
3. From the Build menu, select Build.
4. If you have are using an RC203, you can use file transfer utility provided (FTU2) to download the example to it.

17.2 Agility example: blackbox

Demonstrates:
Interfacing to blackbox modules

Description:
This example shows a top-level `SC_MODULE` with a single `SC_THREAD`. The top-level module instantiates a blackbox module as it would a conventional module, and uses port-mapping in the normal way. The blackbox is defined solely by its ports and its name; it specifically contains no implementation, save a default constructor. Note that the name of blackbox module must correspond to the name of the blackbox core you wish to connect to. The top-level module drives the blackbox with a series of test values. A testbench is provided for simulation that captures the values supplied fed to and returned from the blackbox.

17.2.1 Building the blackbox example for simulation

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the blackbox project and select Set as StartUp Project.
3. Right-click on the blackbox project and select Project Only -> Build Only blackbox

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `blackbox` directory you created:
   ```
   cd ~/agility_examples/blackbox
   ```
3. Build the example:
   ```
   make
   ```
4. Run the example:
   ```
   ./blackbox
   ```

17.2.2 Simulating the blackbox example

The simulation will automatically stop after 5000 simulation cycles. Users can then view the generated waveforms `(blackbox.vcd)` in a suitable waveform viewer.

17.2.3 Building the blackbox example for hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the blackbox project and select Set as Active Project.
3. From the Build menu, select Build.

17.3 Agility Example: reset

Demonstrates:
Using `reset_signal_is` syntax.

Description:
This example shows a top-level `SC_MODULE` with a single `SC_THREAD`. The thread uses the syntax `reset_signal_is` to provide synchronous, thread-level reset behaviour. When the `sc_in` port that the reset is sensitive to satisfies the specified level, the thread is re-started. A testbench is provided for simulation that allows the user to test the reset behaviour.

17.3.1 Building and simulating the reset example

With Microsoft Visual C++

1. Open the solution file `examples.sln` in Visual C++.
2. Right-click on the reset project and select Set as StartUp Project.
3. Right-click on the reset project and select Project Only -> Build Only reset

From the Debug menu, select Start Without Debugging. Select No if prompted to build the other examples.

In Linux

1. Open an X-terminal.
2. Change to the `reset` directory you created:
   ```
   cd ~/agility_examples/reset
   ```
3. Build the example:
   ```
   make
   ```
4. Run the example:
   ```
   ./reset
   ```

17.3.2 Simulating the reset example

When the simulation is started the user will prompted to press enter to increment a counter. At any point, they can enter `r` to reset the thread, or `q` to stop the simulation. The Value Change Dump file (`reset.vcd`) can be viewed in a suitable waveform viewer.

17.3.3 Building the reset example for Hardware

1. Open the workspace `examples.chw` in Agility Compiler.
2. Right-click on the reset project and select Set as Active Project.
3. From the Build menu, select Build.

18 Troubleshooting

My circuit is too large/too slow
Use the results of the logic estimator to pinpoint areas of your code using the most resources.

Look at the application notes and other resources on the Celoxica Web site.

I don't understand the error messages
Look at the ***Error message*** (see page 200) and ***Warning message*** (see page 214) descriptions.

My simulation is not initializing correctly
Delay the clock to invoke the check_clock_start function. (see page 147)

I need more information
Look at the Celoxica technical library at: `http://www.celoxica.com/techlib/`

19 Errors, warnings and information messages

Errors stop a build.

Warnings do not stop a build. The warning level ranges from 1 (severe) to 4 (trivial). They can be suppressed by using the `--warning` ***option*** (see page 184).

Information messages do not stop a build and are always displayed.

Users can create their own ***error, warning and information messages*** (see page 80).

20 Error messages

Most error messages are intuitive. Some of the less obvious ones will be due to system problems, such as files being corrupted, unavailable or in the wrong format, or the system not having enough disk space to write to a file.

C++ error messages are in the range `E5000` to `E5999`. These are not explained further here.

Synthesis error messages are listed below in error number order with a brief explanation.

20.1 Errors E1000 - E1999

E1002	Invalid lvalue in assignment.
E1003	Cannot have unbalanced wait in reset stage (first wait in thread must be balanced).
E1005	No candidate top level module found.
E1008	Function ***P1*** has no body.
E1009	Switch cases must be terminated with a break statement.
E1010	The result of a left shift operation must be cast or assigned immediately to a type of defined width.
E1011	'delete' is not supported for synthesis.
E1012	Values defining a range must be constant.
E1015	Division by zero / occurred during loop unrolling.
E1016	Non-module global variables are not supported for synthesis.
E1017	Array index out of bounds / occurred during loop unrolling.
E1018	Trying to copy a structure containing elements of a type that cannot be copied (sc_signal/sc_fifo/sc_in/sc_out/RAM etc.).
E1019	RAMs/Signals/FIFOs cannot contain elements of type ***P1.***
E1020	Pointer arithmetic is not supported.
E1021	Illegal cast: Type ***P2*** cannot be cast to type ***P1***.
E1022	There is not enough information to determine the value of the loop conditional.
E1023	Command line error: ***P1*** (--help for help).
E1024	No input files specified for compilation.
E1025	Input file: ***P1*** cannot be found.
E1026	No threads or methods found to synthesize.
E1027	Unrecognized file type: ***P1***.
E1028	Thread has been registered without a clock (function ***P1***).
E1029	Port ***P1*** is not mapped and is not top level.
E1030	Range operator's RHS: ***P1*** is greater than or equal to the expression width: ***P2***.
E1031	Range operator's LHS: ***P1*** is greater than or equal to the expression width: ***P2***.
E1032	Range operator's RHS: ***P1*** is less than zero.
E1033	Range operator's LHS: ***P1*** is less than zero.
E1036	Object file not compatible with this version of the compiler.
E1037	Threads can only be sensitive to clocks and not other events.
E1039	Backend synthesis library can't be loaded: ***P1***.
E1040	Backend synthesis library can't be opened: ***P1***.
E1041	For this datatype the range RHS (***P1***) has to be larger than the LHS (***P2***).
E1042	Only one output synthesis type can be specified at a time, use --help for details.
E1043	An output synthesis type must be specified, use --help for details.
E1044	Module has no user-defined constructor.

E1045	Synthesis options are not valid in conjunction with compile-only options, use --help for details.
E1046	Unknown netlist optimization mode, use --help for details.
E1047	EDIF option must have a family and part specified, use --help for details.
E1048	Entry point for synthesis (function 'ag_main') not defined. Nothing to synthesize.
E1049	Virtual inheritance is not yet supported for synthesis.
E1052	Virtual function call on object that cannot be dynamically resolved to an identifier.
E1053	Include directory: ***P1*** cannot be found.
E1056	Output directory ***P1*** cannot be found.
E1057	Error occurred whilst parsing object file - ***P1***.
E1058	Invalid object file, please recompile.
E1059	Cannot read or create object file ***P1***.
E1060	Cannot shift by a negative value.
E1061	Unsupported template argument type.
E1062	Incompatible namespace declarations: ***P1***.
E1063	Multiport index must be resolved to a compile time constant.
E1064	***User-defined error string.***
E1065	An invalid string initializer was encountered.
E1066	Port ***P1*** is already defined on identifier ***P2***.
E1067	Attempting to add a memory port to an identifier that is neither a RAM nor an array of RAMs.
E1068	Attempting to apply 'ag_constrain_ram' to an identifier that is not an array.
E1069	Cannot inherit from built-in data type.
E1070	Remainder of a division by zero / occurred during loop unrolling.
E1077	Library directory: ***P1*** cannot be found.
E1078	Constraint function ***P1*** is called outside ag_main or module constructor.
E1082	Pre-process option cannot be specified with object or library files as input.
E1083	Attempt to derefence an object which does not currently point at an array.
E1084	Calling a function with an ellipsis argument is not supported.
E1085	Dynamically indexed arrays of RAMs are not supported.
E1086	Recursive calls to shared routines are not supported.
E1087	Bit fields are not supported.
E1088	Invalid argument passed to constraint P1 (P2).
E1089	Argument passed to constraint ***P1*** is not a constant (***P2***).
E1090	Type is not supported for synthesis or elaboration.
E1091	Pointer to field constants not currently supported.
E1092	Complex RAM ***P1*** has uninitialized index ***P2*** .
E1093	Only empty case fall-through is currently supported.
E1094	Only a single break at the end of the case stmt is supported.
E1095	Not all elements of array ***P1*** have been ***constrained as RAMs*** (see page 100).
E1096	Number of ***P1*** exceeds specified loop limit.

20.1.1 Error code: E1002

```
Invalid lvalue in assignment.
```

The left hand value is something that cannot be assigned to. For example

```
a.range(3, 5) = x;
```

20.1.2 Error code: E1003

```
Cannot have unbalanced wait in reset stage (first wait in thread must be
balanced).
```

Because all code before the first `wait()` statement is executed on reset, there must be a `wait()` statement in each branch of a fork.

For example

```
if(f>0)
    wait();
else
    return();
```

20.1.3 Error Code: E1005

```
No candidate top level module found.
```

There is not an instance of a class inherited from an `SC_MODULE` in `ag_main`. Agility assumes a single top-level module called from `ag_main`, which in turn calls all subsequent modules, threads and methods.

20.1.4 Error code: E1009

```
Currently switch cases must be terminated with a break statement.
```

The last statement in a `switch` block must be a `break;` No fall-through of cases is allowed.

20.1.5 Error code: E1010

```
The result of a left shift operation must be cast or assigned immediately
to a type of defined width.
```

Since variable widths must be defined by runtime, it is not possible to left shift a variable by an undefined amount, because then the size of its result is unknown.

For example

```
b = 1+ a << x;
```

needs to be

```
b= 1 + (sc_biguint<80>) a << x;
```

or

```
sc_biguint<80> temp = a << x;
b = 1 + temp;
```

20.1.6 Error code: E1011

```
'delete' is not supported for synthesis.
```

Dynamic memory is not synthesizable.

20.1.7 Error code: E1015

```
Division by zero
```

```
Division by zero occurred during loop unrolling.
```

Example 1

```
int i = 0;

int x;

x = x / i;
```

Example 2

Division by zero during loop unrolling:

```
for (i = 5; i != -1; i--)

{

   x = x / i;

   ....

}
```

When the loop body is unrolled, the 6th assignment to `x` is `x = x / 0`.

20.1.8 Error code: E1016

```
Non-module global variables are not supported for synthesis.
```

The only global variables permitted are `SC_MODULE`s.

20.1.9 Error code: E1017

```
Array index out of bounds" / " occurred during loop unrolling
```

This error is produced if an array index is either -1 or it exceeds the dimension of the array.

Example 1

```
int A[10];
int k = 1;
....          // in ... m and k are not changed
m = k-2;
....         //  in ... m is not changed
Result = A[m];
```

Example 2

```
int A[10];
...
A[0] = 5;
for (i=0; i<4; i++)
{
   A[2*i+1] = A[2*i-1] + A[i];
}
```

When this loop is unrolled, the first iteration results in an attempt to look up the -1th item of the array.

Example 3

```
int A[10];
...
A[0] = 5;
for (i=1; i<6; i++)
{
   A[2*i+1] = A[2*i-1] + A[i];
}
```

When this loop is fully unrolled, the final iteration (i.e. when `i`=5) attempts to access the 11th item of `A`.

20.1.10 Error code: E1056

`Output directory` ***P1*** `cannot be found`

You have specified an output filepath that does not exist as a command line parameter. Change the command line or create the missing directory.

20.1.11 Error code: E1063

`Multiport index must be resolved to a compile time constant.`

This occurs if you try to dereference an `sc_port` with a run-time variable

For example:

```
int Index = ReadIn.read();
MyMultiport[Index].write( 4 ); // Error - Index isn't compile-time
resolvable.
```

20.1.12 Error code: E1065

This error occurs if you specify a initialization string that cannot be parsed (e.g. `0x012fg` or `0b120`) or is not a literal.

The initializer must be a literal. i.e. `char*a="121";` is valid, `sc_uint<32> b(a);` is not.

20.1.13 Error code: E1066

`Port` ***P1*** `is already defined on identifier` ***P2***.

This error message is produced if you use `ag_add_ram_port to` add a port to an identifier which already has a port with the same name.

20.1.14 Error code: E1067

```
Attempting to add a memory port to an identifier that is neither a RAM nor
an array of RAMs.
```

This error is produced when you add a port to an identifier using `ag_add_ram_port`, before declaring the identifier as a RAM using `ag_constrain_ram`.

20.1.15 Error code: E1069

```
Cannot inherit from built-in data type
```

This error occurs when you attempt to inherit from a built-in datatype such as sc_uint

20.1.16 Error code: E1070

```
Remainder of division by zero/occurred during loop unrolling
```

Example 1

```
int i = 0;

int x;

x = x % i;
```

Example 2

Attempt to work out the remainder of a division by zero during loop unrolling:

```
for (i = 5; i != -1; i--)
```

```
{
   x = x % i;
   ....
}
```

20.1.17 Error code: E1077

`Library directory:` ***P1*** `cannot be found.`

You have specified a library path (using the `--library-path` or `-L` option in the command-line compiler) which does not exist.

20.1.18 Error code: E1082

`Pre-process option cannot be specified with object or library files as input.`

You have attempted to preprocess an object or library file using the `-E` option in the command-line compiler. For example `agility -E file.ago`.

Error code: E1083

`Attempt to derefence an object which does not currently point at an array.`

You have used a pointer to point at an array element before dereferencing it. Agility only supports pointers to arrays when the pointer is an exact alias of the array.

For instance, the following code:

```
int myArray[10];
int * ptr = myArray[1];
out = ptr[0];
```

needs to be rewritten to this.

```
int myArray[10];
int * ptr = &myArray;
out = ptr[0];
```

Error code: E1085

`Dynamically indexed arrays of RAMs are not supported.`

You are using an unknown value to access an array element. For example

```
int myArray[512][512];
// constrain as 512 elements of 512 RAM
for(int i =0; i<512; i++)
{
    for(int j = 0; j<512; j++)
    {
        myArray[j][i] = j + i; // dimension 'j' is dynamically indexed - we
don't the know values at CT
          wait();
    }
}
```

Error code: E1089
Argument passed to constraint ***P1*** is not a constant (***P2***) .

You have passed a non-constant argument to a constraint function, where a constant is expected.

P1 : the name of the constraint function (e.g. `ag_constrain_clock`)

P2: a description of the parameter (e.g. clock port or signal)

Error code: E1091
`Pointer to field constants not currently supported.`

This may have been generated by something like this:

```
struct foo
{
   int * ptr;
}
typedef int * foo::*PM
```

Is a pointer to data member or pointer to field.

Error code: E1090
`Type is not supported for synthesis or elaboration.`

You have used a non-synthesizable type, such as a union or a bitfield.

Error code: E1092
`Complex RAM` ***P1*** `has uninitialized index` ***P2*** .

When indexing a RAM of complex element type (for example a `struct`), the index can't be uninitialized.

Error code: E1093
`Only empty case fall-through is currently supported.`

C++ allows fall-throughs in a `switch` statement. Fall-through is only allowed in Agility if the `case` statement is empty

Error code: E1094

`Only a single break at the end of the case stmt is supported.`

In a `switch` statement, Agility only permits a single break in the `case` statement, . This break must be at the end of the `case` statement, .

Error code: E1095

`Not all elements of array` ***P1*** `have been constrained as RAMs.`

You cannot partially constrain array elements as RAMs.

For example:

```
int MyRAM[2][2];
ag_constrain_ram( MyRAM[0], ag_synchronous );
```

is not allowed on its own.

20.2 Errors: E2000 -E2999

E2007	Loop has at least one path without a wait and a condition that is not compile-time resolvable.
E2008	Combinational cycle error.
E2009	A break statement is only valid within a switch statement or a loop.
E2010	A continue statement is only valid within a loop.
E2013	Cannot synthesise more than one top level module: ***P1***.
E2015	Module has more than one constructor.
E2017	The target of pointer ***P1*** cannot be statically resolved.
E2018	Pointer dereference cannot be resolved to an identifier.
E2019	The left hand side of this assignment is not a resolvable lvalue.
E2020	Access of structure member ***P1*** on expression that cannot be resolved to an identifier.
E2021	RAMs currently only support integer types.
E2022	Array index target cannot be resolved to an identifier.
E2024	Assignment to ***P1*** in first (reset) cycle is not const.
E2025	The 'goto' statement is not supported for synthesis.
E2027	Thread is declared without a clock.
E2028	Sensitivity declaration with no associated process.
E2029	Identifier ***P1*** is accessed in multiple threads.
E2030	Assignments as expressions are not yet supported.
E2031	Runtime loops are not supported in the reset cycle.
E2032	Runtime conditions are not supported in the reset cycle.
E2033	Channel accesses are not supported in the reset cycle.
E2034	Port reads are not supported in the reset cycle.
E2040	Ports cannot be written to from more than one clock domain.
E2041	Two top level input ports cannot be mapped together (***P1*** ***P2***).
E2042	The length of a FIFO must be a constant.
E2043	The address value of a pointer cannot be accessed.
E2045	FIFO accesses are not supported in the reset cycle.
E2046	SystemC sc_uint or sc_int type cannot be wider than 64 bits. Consider sc_biguint or sc_bigint types instead.
E2047	SystemC types must be at least one bit wide.
E2048	Arithmetic operations are not permitted on bit vectors. Consider using integer types instead.
E2049	One or more parameters to built-in function ***P1*** could not be resolved to a compile-time value.
E2050	Function ***P1*** declared as a built-in function was not found. Possible obsolete or invalid.
E2051	Built-in function ***P1*** called with wrong number or type of arguments. Possible obsolete or invalid header file.
E2052	sc_inout ports can only be used with type sc_lv.

E2056	The result of a range operation must be cast or assigned immediately to a type of defined width.
E2057	Cannot map port ***P1*** on top level module.
E2058	Ports cannot be mapped to multiple entities.
E2059	Non-blocking fifo accesses are not yet supported.
E2060	sc_inout ports (***P1***) are not supported for synthesis in black boxes.
E2061	***P1*** not allowed in a process.
E2062	Multiple definitions of variable ***P1*** found.
E2063	Multiple definitions of routine ***P1*** found.
E2064	Routines differ only on return type (***P1***).
E2065	Type mismatch between declarations of variable ***P1***.
E2066	Synchronous and asynchronous assignments are not permitted in the same sc_method.
E2067	Synchronous assignments in different clock contexts are not permitted in the same sc_method.
E2068	Incomplete asynchronous assignment to ***P1*** in sc_method.
E2069	Variable ***P1*** read before written in asynchronous context in sc_method.
E2070	Wait is not permitted in sc_method.
E2071	Identifier ***P1*** missing from sensitivity list in sc_method.
E2072	Global asynchronous reset can only be declared in the top level module chosen for synthesis.
E2073	This design already has a global asynchronous reset declared.
E2074	This thread has already been declared sensitive to a reset.
E2075	Resource limit with no current thread declared.-
E2076	Memory ***P1*** does not have enough ***P2*** ports.
E2078	Sensitivity for sc_spawn_options must match sensitivity for parent thread.
E2079	Sensitivity for sc_spawn_options is NULL.
E2080	CThreads using SC_FORK/SC_JOIN must not specify a reset signal.
E2081	Unresolved identifier ***P1***.
E2082	Identifier ***P1*** is initialised in both design elaboration and thread execution.
E2083	Reset declaration with no associated clocked thread process (SC_CTHREAD).
E2084	"Non-void function with no explicit return."
E2085	P1 << " are not supported inside a thread."
E2086	Incomplete element write of RAM ***P1*** at address ***P2*** .
E2087	"The type of this expression is only supported for compile time evaluation not for synthesis."
E2088	Continue statement not supported in 'for' loops.
E2089	Array size is not compile-time determinable.
E2090	Multiple calls to shared routine ***P1*** in the same clock cycle are not allowed, as this routine may take zero cycles to execute
E2091	Agility does not support the ***ternary operator*** (see page 67) on the left hand side of an assignment
E2092	Ranges must be cast to the target type when used in conditional expressions.

E2093	Bit select operator must be cast to the target type when used in conditional expressions.
E2094	Users must call .read() on ports and signals when used in conditional expressions.

20.2.1 Error code: E2017

`The target of pointer` ***P1*** `cannot be statically resolved.`

It is not possible to determine at compile time what a pointer points at).

Example

```
sc_uint<16> *p;
....
if ( Val.read() % 2 == 0)
{
p = &x;
}
else
{
p = &y;
}
z = *p;
....
```

`p` points to `x` if the value read from `Val` is even and to `y` otherwise. The value read from `Val` is not known at compile time, and so the compiler does not know whether `p` points to `x` or `y`. Attempting to resolve the pointer (i.e. `z = *p;`) causes Agility to report an error message.

`Memory` ***P1*** `does not have enough` ***P2*** `ports.`

This error message is produced during synthesis if a memory does not have enough read or write ports to perform the required accesses.

20.2.2 Error code: E2081

Unresolved external reference ***P1***.

The definition of ***P1*** cannot be found. Check if you have failed to link in any objects or library files.

20.2.3 Error code: E2082

`Identifier` ***P1*** `is initialized in both design elaboration and thread execution.`

You have initialized a variable in both global scope and inside a thread:

Example

```
#include "systemc.h"
sc_uint<32> a = 10; // global initialiser
SC_MODULE( myModule )
{
    sc_in<bool> Clock;
    sc_out<sc_uint<32> > Result;

public:
    void myThread()
    {
        a = 20; // thread initialiser - error
        wait();

        while ( 1 )
        {
            Result = a;
            wait();
        }
    }

    SC_CTOR( myModule )
    {
        SC_THREAD( myThread );
        sensitive_pos << Clock;
    }
};

void ag_main()
{
    myModule Top( "top" );
}
```

Error code: E2083

`Reset declaration with no associated clocked thread process (SC_CTHREAD).`

You have called `reset_signal_is` on a thread that is not a clocked thread.

Error code: E2090

`Multiple calls to shared routine` ***P1*** `in the same clock cycle are not allowed, as this routine may take zero cycles to execute.`

Agility does not support multiple calls in the same clock cycle to a shared function that may take zero clock cycles to execute.

Error code: E2092

`Ranges must be cast to the target type when used in conditional expressions.`

If users use `.range(..)` on a variable in either branch of a (`?:`) operator they must also cast to the target type (i.e. the type on the LHS of the assignment).

Error code: E2093

```
Ranges must be cast to the target type when used in conditional
expressions.

Bit select operator must be cast to the target type when used in
conditional expressions.
```

If users use `[ ]` on a variable in either branch of a (`?:`) operator they must also cast to the target type (i.e. the type on the LHS of the assignment).

Error code: E2094

```
Users must call .read() on ports and signals when used in conditional
expressions
```

This error may have been generated by code such as this:

```
sc_signal<int> s1, s2;
...
int foo = (Port.read() ? s1 : s2); // .read() has not been called on signal
s1
```

21 Warning messages

Most warning messages are relatively intuitive. Some of the less obvious ones are listed in number order with a brief explanation. Some of the error messages are also described in the Agility User Guide.

C++ warning messages are in the range `W6000` to `W6999`. These are not explained further here.

- Warnings: W3000 - W3999
- Information messages: I4000 - I4999

21.1 Warnings: W3000 - W3999

Number	*Level*	*Message*
W3001	2	Variable ***P1*** can be read before it is written.
W3004	2	Number of places to shift by could be negative.
W3005	3	Output port ***P1*** is never used.
W3006	3	Input port ***P1*** is never used.
W3007	3	In/Out port ***P1*** is never used.
W3008	1	***P1*** are not supported in a module constructor.
W3010	1	Port/Channel/RAM ***P1*** written to from multiple threads. Undefined behaviour may result.
W3011	2	RAM initialization should happen in the constructor of a module only.
W3012	3	FIFO ***P1*** is never written to.
W3013	1	Output port ***P1*** is not written to on every clock cycle .
W3014	2	SystemC sc_int or sc_uint intermediate value exceeded 64 bits with possible loss of data. Consider using sc_biguint sc_bigint or casting operands to smaller types.
W3016	2	Requested too few instances of resource for thread (needed ***P1***).
W3017	2	Requested too many instances of resource for thread (needed ***P1***).
W3018	1	Include path: ***P1*** cannot be found.
W3019	3	Invalid constraint specified for object; ignored.
W3020	2	Variable ***P1*** may be read before being initialized.
W3022	2	***User-defined warning string.***
W3023	3	RAM identifier ***P1*** has no ports specified, assuming single port read-write access.
W3024	2	"Port/Signal '" << P1 << "' cannot be traced as it is not implemented as a register."

22 Information messages: I4000 - I4999

I4001	Top level module '***P1***' found for synthesis.
I4002	Loop was unrolled with '***P1***' iterations.
I4004	CDFG Output file generated.
I4006	***User-defined information message string.***

23 Troubleshooting: FIFOs

My FIFOs do not run at the required clock frequency
You could

- read/write directly from/to a register
- use a block RAM rather than a LUT RAM or SelectRAM
- choose the size of the FIFO so it only uses one memory block

FIFOs seem to have erratic timing
FIFOs use a different implementation if their width is a power of 2.

Trouble-shooting if Quartus cannot use a pin
If Quartus complains about being unable to use a pin (e.g. H19) then use the setting below in the Quartus GUI.

1. Select Device from the Assignments menu.
2. Click the Device & Pin options button.
3. Select the Dual purpose pins tab.
4. Choose Use as regular I/O for Data 0 (and/or possibly for the other options as well).

24 Glossary

A

ASIC
Application Specific Integrated Circuit

B

Block RAM
Specialized embedded memory modules within an FPGA providing higher density storage than distributed RAM. May provide multiple ports (depends on device type). Block RAMs on Altera Apex devices are called ESBs. Block RAMs on Altera Stratix devices are called tri-matrix blocks.

Blocking
A read or write is blocking if it blocks execution until it has completed successfully. For example, a FIFO read may block until there is data in the FIFO to read.

Buffer
Register or segment of memory used to store data temporarily. Often used to pass data between devices or processes that operate at different speeds.

C

C++
An object-oriented programming language developed by Bjarne Stroustrup and based on the C programming language.

CDFG
Control and Data Flow Graph. A graphical representation of a design, consisting of hierarchical blocks (also known as groups), nodes and edges (also known as arcs).

Clock net
Wire that distributes a clock round an FPGA to all components that require it.

Combinational loop
Code that generates a hardware loop consisting of pure logic, with no latches or delays.

Combo box
A drop-down list in an IDE window.

Compile-time determinable
A value that can be determined when the code is compiled, before it has run.

Compile-time evaluation
The process of discovering whether a value can be determined, and is so, deciding what value it is, carried out at the compilation stage.

CPLD
Complex Programmable Logic Device. Programmable logic chip supporting thousands or hundreds of thousands of gates. Consists of arrays of logic blocks with programmable interconnections, surrounded by programmable I/O

CSE
Common Sub-expression Elimination. If you select the CSE option for optimizing your code, the compiler tries to identify similar expressions and specifies that they share resources.

Cthread process
A thread process which is associated with a specified clock, and which may be reset.

D

Distributed RAM
LUTs used as RAM on a device. Also see Block RAM.

DRAM
Dynamic Random Access Memory. Memory is dynamic in that the storage cells need to be refreshed periodically. Requires power to maintain content. Also see SDRAM.

E

EAB
Embedded Array Block. Type of block RAM found on Altera Flex devices.

EDIF
Electrical Design Interchange Format. Standard format for representing electronic designs.

F

FIR filter
Finite Impulse Response filter. A filter that uses convolution rather than recursion.

Fixed-point
A method of storing numbers with a fixed number of digits after the decimal point.

Flip-flop
Circuit that alternates between two states (0 and 1). The flip-flop maintains its state until an input is received.

Floating-point
A manner of representing real numbers on a computer, by storing them in two parts: the integer (significand or mantissa) and its exponent. They are stored to a specified precision

FPGA
Field Programmable Gate Array.

G

Gate
Low level digital component. Logic gates perform Boolean functions (AND, OR etc.).

H

HDL

HDL
Hardware Description Language, e.g. VHDL, Verilog.

Header file
File containing function and macro prototypes. Enables functions to be shared between files. You must `#include` header files in each source file that uses them. Header files have a sc.cpp extension in SystemC.

I

IDE
Integrated Development Environment. A graphical user interface

Integer
A whole number without a fractional part. In SystemC, an integer can be of any specified width.

L

LUT
Look Up Table implemented in hardware as multiplexers with SRAM

M

MDI
Multiple Document Interface

Method process
A process which terminates and does not contain wait statements. It may be initiated by any signal that it is sensitive to.

Module
The basic building block of SystemC. Modules may contain other modules, logic, processes, and ports.

Module hierarchy
The way modules may inherit, or the way they may contain sub-modules.

O

OSCI
Open SystemC Initiative. The group who define and develop the SystemC standard. Further details available from

http://www.systemc.org
(http://www.systemc.org)

OSCI simulator
The publicly available SystemC simulator provided by OSCI.

P

Port
The point where one module or process is connected to another.

Port mapping
The manner in which one module or process is connected to another.

Q

Quantization mode
The means used to quantize a value for use in fixed-point representation.

S

Sensitivity
The effect of making a thread or method process sensitive to one or more clocks or signals. If a signal is received, the process starts or resumes.

SystemC
A system description language implemented in C++, which has been designed to enable users to use the same language throughout the development of a circuit.

T

Thread process
A process which runs in parallel with other thread processes and normally does not terminate. It may contain `wait` statements which suspend the process until a signal has been received

V

VHDL
Very high speed integrated circuit Hardware Description Language.

W

Wrapping
The procedure of continuing to write into the least significant bits after the most significant bits have been filled

Additional resources

You can find further information on the Celoxica Web site.

KnowledgeBase	Available from `http://www.celoxica.com/support/`. You will need your support password.
Technical library	`http://www.celoxica.com/techlib/` Contains application notes, white papers, FAQs, data sheets etc.

24.1 Legal notices

The Software is a development tool. As a result the use to which the Software is put and the designs developed using the Software are outside the control of Celoxica.

Use of the Software is conditional upon your entering into an End User license.

Celoxica has not tested the Software in safety critical environments including control of aircraft, air or surface traffic, aircraft navigation, or aircraft communications, medical applications, the design, construction, operation or maintenance of any nuclear facility, national defence or any other activity involving high risk or life critical procedures or activities.

Copyright © 1991 - 2006 Celoxica Limited. Celoxica and the Celoxica logo are the trademarks of Celoxica Limited. All other products or services mentioned herein may be trademarks of their respective owners.

24.2 Agility licenses

If you do not have a full Agility license, you may be restricted in the type of output you can generate (e.g. EDIF, VHDL and Verilog may not be available). Your license may also restrict the devices you can generate code for, or your ability to use SystemC interfaces. Licenses are also available for various SystemC libraries, such as those targeting Celoxica's reconfigurable platforms.

If you do not have a license to use interfaces you can only target interfaces within any libraries you hold licenses for. You do this by using the macros provided in the libraries. For example, you might use a PAL macro to target external memory on a board.

Agility library licenses are generated using RSA Data Security, Inc. MD5 Message-Digest Algorithm.

25 Index

- operation 65

.

.gcf 151

A

Actel 128, 151
  optimizing arithmetic hardware 128
Actel Designer 151
ag_add_ram_port 103, 104, 105
ag_assert_known 79
ag_assert_known_and_true 79
ag_assert_unknown_or_true 80
ag_constrain_clock 92, 93
ag_constrain_ram 100, 102
ag_constrain_resource 112
ag_error 80
ag_foreign_module 72, 73
ag_global_async_reset_is 74
ag_information 80
ag_main 90, 91
ag_override_global_init 76
ag_override_global_inits 77
ag_resource_addsumcmp 112
ag_resource_multiplier 112
ag_resource_type 112
ag_undefined 75
AG_UNDEFINED 75
ag_warning 80
Agility messages 200
agility.h 72
--agility-library-file option 184
--agility-object-file option 184
Altera 128
ALU mapping 128
--alu-limit 128
--alu-limit option 186
--alu-mapping option 186
area 142, 143
  estimating 141, 142, 143
  reducing 141
arithmetic hardware in Actel devices 128
arrays 54
assertions 78
asynchronous memory 100
automatic tree balancing 125

B

bigint 60
birdseye window 174
bit fields 59
bit vector 64
black box
  RAMs 107, 108
black boxes 72
block RAM 100
bool 52
build 19
  EDIF 21
  SystemC 19
  viewing reports 22

C

-c option 184
C++ datatypes 52
C++ floating-point types 53
C++ integer types 52
CDFG 166, 167
  creating 173
  display options 171
  edges 168, 171
  groups 167, 168
  linking to source code 175, 177
  nodes 167, 169
  overview 166
  viewing 173, 177
  viewing shared operations 175
celoxica.lmf 150
celoxica.v library 151
celoxica.vhd library 152
CELOXICA_OSCI_SIM 165
channels 42

FIFOs 43
signals 42
signals with default 80
char 52
check_clock_start 147
classes 57
clock 44, 47, 49
clock constraints 92, 93
example 93
code depth 141
combinational adder example 48
combinational paths 142, 144
estimation 142, 144
longest 142, 144
command-line 183
options 183
common sub-expression elimination 124
compilation reports 144, 145
--compile-only option 184
compile-time determinable 55, 56
compile-time evaluation 46, 54
compile-time recursion 118
conditional rewriting 125
conditional statements 69
constraints 91, 150, 151
clock 92, 93
FIFO 93
output 114
RAM 100
resource 112, 113
synthesis 92
timing 150, 151
control-flow 170
conventions 10, 11
conversion 65
creating 14
creating a CDFG file 173
CSE 124

D

-D option 183
data flow 169
datatypes 52
bools 52
chars 52
classes 57
doubles 53
floats 53
ints 52
long doubles 53
long ints 52
long long ints 52
sc_bigint 60
sc_biguint 60
sc_bit 64
sc_bv 64
sc_int 60
sc_logic 64
sc_lv 64
sc_uint 60
structs 57
summary 65
synthesizable 52
default value for signals 80
delay 141, 142, 144
estimating 141, 142, 144
delaying simulation clock 147
delete operator 68
depth 141, 142, 144
estimating 142, 144
reducing 141
design flow 90
development sequence 13
different clocks 93
distributed RAM 99
double 53

E

-E option 184
edges 168, 171, 175
viewing 171, 172, 176
--edif 184
EDIF 130, 131, 150
place and route tools 150

setting up Quartus 150
standard 130
Enable retimer 129
error messages 200, 214
warnings 214
errors 80, 200
--estimate-logic-size option 185
examples 73, 93
arrays 54
assertions 79
black box 73
black box dual-port RAM 108
C++ floating-point types 53
C++ integer types 52
clock constraints 93
combinational adder 48
default signal 82
dual-port RAM 105
Fibonacci series 118
FIFO constraints 99
global asynchronous rest 74
half-adder 33
initialization 75, 76, 77
logic area and depth estimation 143, 144
named mapping 40
output constraints 114
pipelining at output dual-port RAM 111
pipelining at output: 110
pointers 55
process declaration 44
RAM 102
range 60
resource constraints 113
ROMs 104
SC_CTHREAD 47
SC_CTOR 33
sc_module 35
sc_spawn 51
SC_THREAD 46
shift 60
synchronous adder 49
thread process 46
tri-state ports 39
virtual inheritance 58
exceptions 71
Expand netlist for option 128

F

-f option 186
--family option 186
FAQ 220
--fast-carry-chains 186
FF 143
Fibonacci series example 118
FIFO constraints 93
example 99
metastability_protection_level 98
fifo_timing 93
FIFOs 43
fine-grained logic sharing 125
fixed-point types 62
flip-flops 143
float 53
floating-point 53
foreign modules 72
function call hierarchy 176
functions 70, 119
inlining 119
virtual 57

G

GCF files 151
Generate estimation information 141
getting started 14
global synchronous reset 74
groups 168

H

-h option 183
hardware 147
targeting 147, 150, 151, 152
HDLs 150
VHDL 152
--hdl-style option 186

help 200, 220
--help option 183
hierarchical modules 35
high-level synthesis 120

I

-I option 183
if statements 69
--include-path 183
information 220
information messages 80
inheritance 35
initialization 75, 76
  thread processes 46, 76
Input ports 37
Input/output ports 38
instantiating foreign code 72
int 52
integer 52, 60

K

KnowledgeBase 220

L

left shift operation 60
libraries 85
  celoxica.v 151
  celoxica.vhd 152
library mapping file 150
licenses 220
lmf 150, 151
logic area 141, 142, 143
logic depth 141, 142, 144
logic estimator 141
logic sharing 123
long double 53
long int 52
long long int 52
look-up tables 143
loop unrolling 116
  limitations 117
loops 70
LUTs 143

M

mapping
  mapped ALUs 128
  synchronous or asynchronous 100
MATLAB 220
messages 200
metastability 93
method processes 48
minus operation 65
module 32, 33, 35
  foreign 72, 73
  hierarchical 35
Module Call Graph 176
multiple clocks 93
multi-port RAMs 104

N

-N option 128
named mapping 40, 45
namespaces 71
NAND gates 143
neg() 49
negedge 49
--netlist-opt option 186
new 37, 55, 68
node types 177
--no-default-include 183
nodes 169
--no-hierarchy option 186
numeric types 59

O

-o option 184
operators 67
  - operation 65
  new 55, 68
optimizations 124
  conditional rewriting (CR) 125
  fine-grained logic sharing 123
  rewriting 125
  tree balancing 125
optimizer 141

optimizing code 128, 141
arithmetic hardware in Actel devices 128
ouput names of SystemC threads 134, 138
output constraints 114
example 114
stages 115
output formats 130
EDIF 130
RTL SystemC 139
Verilog 135
Verilog-2001 135
VHDL 132
output names 134, 138
Output ports 38
--output-cdfg option 185
--output-file-option 184
overflow mode 62

P

-p option 186
--part option 185
--pipeline-ram option 186
pipelining 109
at the output 109
place and route tools 150
pointers 55
ports 92
register inference 120
Ports 37
I/O 38
input 37
mapping 40
named mapping 40
output 38
read/write operations 40
tristate 39
user-defined 40
--ports-as-pins 186
pos() 49
posedge 49
--preprocess option 184
preprocessor
define 183
undefine 183
--preprocessor-define 183
--preprocessor-undefine option 183
ProASIC devices 128, 151
constraints files 151
optimizing arithmetic 128
processes 43, 45, 48
methods 48
restrictions 48
sensitivity 44
spawning 50
threads 45
project 14
project development sequence 13
projects 13
developing 13
promotion 65

Q

quantization mode 62
Quartus 150

R

RAMs 100
black box 107
block 99
constraints 100
distributed 100
example 102, 104, 110, 111
multi-port 104, 105
pipelining 109
ports 103
restrictions 101
versus arrays 100
range operator restrictions 60
read 40
recursive definitions 118
reducing logic depth and area 141
references 56
register inference 120
--repeated-conditional-rewriting option 188
report file 144, 145

report files 144
--report option 185
reports 144
reset 45, 47, 74
reset_signal_is 47, 74
resource constraints 112, 113
resources 220
restrictions 55
  arrays 54
  left shift 60
  new 55
  pointer 55
  range 60
  references 56
  virtual functions 57
resume 45
retiming 75, 129
--retiming 186
reviewing synthesis 141
rewriting optimization 125
ROMs 103
rounding 62
RTL SystemC 147

S

saturation 62
sc_bigint 60
sc_biguint 60
sc_bit 64
sc_bv 64
sc_clock 147
SC_CTHREAD 45, 47
SC_CTOR 33
sc_fifo 43
sc_fixed 62
SC_FORK 50
SC_HAS_PROCESS 34
sc_in 37
sc_in_clk 37
sc_in_resolved 37
sc_in_rv 37
sc_inout_clk 38
sc_inout_rv 39
sc_int 60
SC_JOIN 50
sc_logic 64
sc_lv 64
sc_main 90
SC_METHOD 49
sc_module 32, 35
SC_MODULE 33
sc_out 38
sc_out_clk 38
sc_out_resolved 38
sc_out_rv 38
sc_port 40
SC_RND (rounding to plus infinity) 62
SC_RND_CNV (convergent rounding) 62
SC_RND_INF (rounding to infinity) 62
SC_RND_MIN_INF (rounding to minus infinity) 62
SC_RND_ZERO (rounding to 0) 62
SC_SAT (saturation) 62
SC_SAT_SYM (symmetrical saturation) 62
SC_SAT_ZERO (saturaton to 0) 62
sc_signal 42
sc_spawn 50, 51
  restrictions 51, 52
  sc_spawn_options 50, 52
SC_THREAD 45, 46
SC_TRN (truncation) 62
SC_TRN_ZERO (truncation to 0) 62
sc_ufixed 62
sc_uint 60
SC_WRAP (wrap-around) 62
SC_WRAP_SM (sign magnitude wrap-around) 62
sensitive 33, 46
sensitivity 33, 44, 46
shared logic 123
sharing 123
short 52
sign magnitude wrap-around 62

signal 42
with default 80
signals..... 92
register inference 120
signed 52
simulating your design..... 164
simulation..... 147
spawning threads 50
statements 69
do - while 70
for 70
if 69
switch 69
while 70
static pointer resolution..... 119
structs 57
summary.html 142
suspend 45
switch statements 69
synchronous adder example 49
synchronous memory 100
synthesis..... 90, 115
constraints 92
reviewing results 141
synthesis guide 90
workflow 90, 115
synthesis entry point..... 18
synthesizable subset 32
synthesizing pointers, arrays and references 54, 55, 56
--systemc option 184
SystemC types 59
--systemc-source-file option 184

T

target..... 147
targeting hardware..... 147, 150, 151, 152
synthesis tools 123
via EDIF 150
via Verilog 151
via VHDL 152
Tcl files..... 150
technical papers..... 220
--technology-mapping 186
thread processes 45, 46, 47
initalization 46
timing 150, 151
constraints 150, 151
timing constraints..... 92, 93, 129
--timing-constraints 186
tree balancing 125
--tree-balancing option..... 188
tristate ports 39
troubleshooting 200
truncation 62
tutorial 14
tutorials 220
type conversion..... 65
type promotion 65
types 52

U

-U option 183
unions 59
unsigned 52

V

-v 183
Verilog..... 135, 151
hierarchy 136
module names 136
no hierarchy 136
port and wire names 139
standard 135
threads 138
Verilog-2001 135
Verilog 2001..... 152
--verilog option 184
--verilog-2001 option..... 152, 184
--version 183
VHDL 132, 150, 152
entity naming 132, 134
file names 134
hierarchy 133
ports 134, 135

standard 132
threads 134
--vhdl option 184
viewing 22
build reports 22
CDFG file 19
viewing a CDFG file 173, 177
virtual functions 57
virtual inheritance 35, 58

W

-W compiler option 184
wait statements 45
wait(n) 45
wait_until 45
walk-through 14
warning messages 200, 214
--warning-level option 184
warnings 80, 214
white papers 220
workflow 14
wrap 65
wrap-around 62
write 40

X

Xilinx 128